WL-TR-95-1119

**AVIONICS SOFTWARE REENGINEERING
TECHNOLOGY (ASRET) PROJECT,** VOLUME 1:
Project Summary, Account, and Results

**D.E. WILKENING
J.P. LOYALL**

TASC
55 Walkers Brook Drive
Reading, Massachusetts 01867

19960319 006
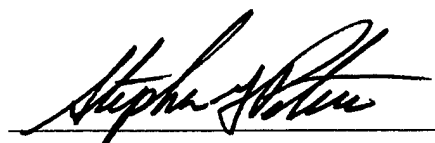
DTIC QUALITY INSPECTED 1

NOTICE

When Government drawings, specifications or other data are used for any purpose other than in connection with a definitely Government-related procurement, the United States Government incurs no responsibility or any obligation whatsoever. The fact that the government may have formulated or in any way supplied the said drawings, specifications, or other data, is not to be regarded by implication, or otherwise in any manner construed, as licensing the holder, or any other person or corporation; or as conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

This report is releasable to the National Technical Information Service (NTIS). At NTIS, it will be available to the general public, including foreign nations.

This technical report has been reviewed and is approved for publication.

KENNETH LITTLEJOHN, Project Engineer
Software Concepts Section
WL/AAAF-2

WILLIAM R. BAKER, Acting Chief
Avionics Logistics Branch
WL/AAAF

STEPHEN G. PETERS, Lt Col, USAF
Deputy Chief
System Avionics Division
Avionics Directorate

If your address has changed, if you wish to be removed from our mailing list, or if the addressee is no longer employed by your organization, please notify WL/AAAF, WPAFB, OH 45433-7301 to help us maintain a current mailing list.

Copies of this report should not be returned unless return is required by security considerations, contractual obligations, or notice on a specific document.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (*Leave blank*)	2. REPORT DATE May 1995	3. REPORT TYPE AND DATES COVERED Final 5/1/92 – 5/1/95

4. TITLE AND SUBTITLE
Avionics Software Reengineering Technology (ASRET) Project
Volume I Project Summary, Account and Results

6. AUTHOR(S)
D.E. Wilkening, J.P. Loyall (TASC)

5. FUNDING NUMBERS
C F33615-92-D-1052
PE   78012 F
PR   3090
TA   01
WU   14

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
TASC, Inc.
55 Walkers Brook Drive
Reading, MA 01867

8. PERFORMING ORGANIZATION REPORT NUMBER
TASC: TR-06661-4

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Avionic Directorate
Wright Laboratory
Air Force Materiel Command
Wright Patterson AFB OH 45433-7409

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
WL-TR-95-1119

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (*Maximum 200 words*)

The objective of the Avionics Software Technology Support (ASTS) program is to perform research and development for enhancing Embedded Computer System (ECS) software development and post-deployment support. The Avionics Software Reengineering Technology (ASRET) project is the second delivery order under ASTS. Under ASRET, we investigated existing reengineering and reverse engineering process, techniques, and software tools. Based upon this study, we developed a process model and environment for reengineering software from one language (FORTRAN) to another (Ada). We designed and implemented a Reengineering Tool (RET) prototype to assist the engineer in this process. We evaluated the RET by translating FORTRAN simulation code for Block 40 of the F-16 OFP to Ada. To prove the value of the RET, we recommend that software maintainers participate in an experiment using an enhanced RET to reengineer an application in a production environment rather than in a laboratory.

14. SUBJECT TERMS
Reengineering, Reverse Engineering, Reuse

15. NUMBER OF PAGES
82

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL

NSN 7540-01-280-5500

STANDARD FORM 298 (Rev. 2-89)
Prescribed by ANSI Std 239-18
298-102

MI009
1-22-92

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

EXECUTIVE SUMMARY

The Avionics Software Technology Support (ASTS) program is an ongoing activity of the Software Concepts Group, Avionics Logistics Branch at Wright Laboratory (WL/ AAAF-3), Wright Patterson Air Force Base, Ohio. The objective of the ASTS program is to perform research and development for enhancing Embedded Computer System (ECS) software development and postdeployment support.

The Avionics Software Reengineering Technology (ASRET) project is the second delivery order (DO) under ASTS. This document is the final report for the ASRET project which concluded on 1 May 1995. The ASTS program continues beyond ASRET with several active DOs which are treating complementary research topics.

Under ASRET, we investigated existing reengineering and reverse engineering processes, techniques, and software tools. Based upon this study, we developed a process model and environment for reengineering software from one language (FORTRAN) to another (Ada). The approach is to engineer an Ada program by reusing portions of the original FORTRAN design and implementation. We designed and implemented a Reengineering Tool (RET) prototype to assist the engineer in this process. While the process model was developed to treat the specific FORTRAN-to-Ada reengineering, it is easily extensible to other source/target language combinations.

The RET applies reverse engineering techniques that facilitate redocumenting and recovering the design of a legacy system to help the engineer inspect and understand the system. The RET's restructuring capabilities help the engineer develop a program using design and implementation information recovered from the subject system.

The RET automatically translates low-level program statements, relieving the engineer from the tedious syntactical minutia, i.e., differences between the source and target programming language syntax, that divert attention from the more important design and implementation decisions requiring human judgement. By concentrating on tasks that are well-suited to automated support, the RET reduces the resources needed to reengineer avionics support software and helps the human engineer produce a more maintainable system.

We evaluated the RET prototype by translating FORTRAN source code, simulation software for Block 40 of the F-16 Operational Flight Program (OFP), to Ada. We are convinced after exercising the RET that the ASRET process model is sound and the top-down

reengineering style that it encourages is effective. One of our goals for the RET prototype was that it help the engineer capture as much of the existing design from the original program as possible, and we accomplished this with the RET.

The RET is a prototype that we developed to evaluate reengineering technology and we recommend certain improvements that would allow production software maintainers who are not experts in language processing to achieve acceptable productivity. To prove the value of the RET, we recommend that software maintainers participate in an experiment, i.e., a Beta test, using an enhanced RET to reengineer an application in a production environment rather than in a laboratory.

1.   INTRODUCTION

This report presents the findings of the Avionics Software Reengineering Technology (ASRET) project. The ASRET project is the second DO issued to TASC under the Avionics Software Technology Support (ASTS) program.

1.1   BACKGROUND

Context — The ASTS program is an ongoing activity of the Software Concepts Group, Avionics Logistics Branch at Wright Laboratory (WL/AAAF-3), Wright Patterson Air Force Base, Ohio. The objective of the ASTS program is to perform research and development for enhancing Embedded Computer System (ECS) software development and postdeployment support. The main objective of ASRET was to develop an automated Reengineering Tool (RET) prototype for avionics support software.

The reengineering of software from one language to another is becoming a necessity as Air Force organizations strive to modernize and improve the maintainability of their systems while avoiding the excessive costs of new development. Systems that have been in use for years often incur large maintenance costs (Ref. 1) for a number of reasons.

- Continual maintenance has made the current implementation and original design inconsistent, the code harder to understand and error-prone, and the documentation out-of-date.

- They are written in languages that have fallen out of favor. The limited selection of support tools for these languages, the corresponding expense of these tools, and the shrinking pool of qualified programmers to maintain the software adds to the expense of maintenance.

- They were developed without modern software engineering practices, resulting in code that lacks structure and is difficult to understand.

- Employee turnover has reduced the staff's understanding and "intimate" knowledge of the system.

Wright Laboratory initiated the ASRET project to help reduce maintenance costs for legacy systems and to assist in the evolution to Ada. To this end, we developed an environment for reengineering software from one language to another. We concentrated on the reengineering of avionics simulation software written in FORTRAN to Ada, and designed the RET so that it could support additional languages in the future.

1.2 TASK STRUCTURE

The objective of the ASRET project was to develop an automated Reengineering Tool (RET) prototype for avionics support software. The specific goals included investigating existing reengineering and reverse engineering processes, techniques, and software tools, defining a reengineering process model, and building a RET prototype that supports

- Translating avionics simulation software written in FORTRAN to Ada,
- Improving the software through restructuring techniques,
- Changing the design of the software so that it is consistent with modern software engineering principles, and
- Adding documentation that is consistent with the software.

The ASRET project task structure is shown in Table 1.

Table 1   ASRET Task Structure

ASRET TASK	
NUMBER	NAME
1	Software Reengineering Study
2	Reengineering Process Model Development
3	Reengineering Tool Development
4	Reengineering Tool Testing and Evaluation

For Task 1, *Software Reengineering Study*, we conducted an extensive investigation of software reengineering tools and methods. The goal was to collect, organize, and present information on software reengineering tools and methods that might be relevant to ASRET, and to record the information for use in the subsequent tasks. The results of Task1 are documented in the Software Reengineering Study Report (Ref. 2).

During Task 2, *Reengineering Process Model Development*, we developed a reengineering process model based upon the results of Task 1, and developed the Software Requirements Specification (SRS) (Ref. 3) for the Reengineering Tool (RET) prototype.

In Task 3, *Reengineering Tool Development*, we designed (Ref. 4) and implemented the RET prototype and exercised the RET by transforming the Fire Control Radar (FCR) subsystem of the F-16 Block 40 OFP simulation software provided to us by Wright Laboratory for this purpose.

We executed the RET prototype and converted most of the other subsystems in the Block 40 code to Ada in Task 4, *Reengineering Tool Testing and Evaluation*. The output that we generated appears in Volume II (Ref. 5).

1.3 ACCOMPLISHMENTS

This section summarizes the activities that we performed under, and the results of the four ASRET tasks shown in Table 1.

Software Reengineering Study

For Task1, Software Reengineering Study, we performed an extensive investigation of existing reengineering and reverse engineering processes, techniques, and tools. The study comprised two parts: the *Literature Survey*, and the *Software Survey*. We documented the results of the study in the Task 1 report, entitled "Avionics Software Reengineering Technology (ASRET) Software Reengineering Study Report" (Ref. 2).

Reengineering Process Model Development

Based upon the results of the Software Reengineering Study that we conducted in Task 1, we developed a process model that defines reengineering in terms of (nondestructively) engineering a new program by reusing the design and implementation of the original program. The process model is consistent with well-accepted reengineering models (Refs. 6, 7), and improves on them by dividing automated restructuring from restructuring that requires human insight and by defining restructuring tasks in terms of modern software engineering practices.

The ASRET Reengineering Process Model is documented in the "Software Requirements Specification for the Avionics Software Reengineering Tool (RET) Prototype System RET-SRS-01" (Ref. 3). Section 3 summarizes the process model and includes various refinements to the model that we made during Tasks 3 and 4.

Reengineering Tool Development

We designed and implemented a Reengineering Tool (RET) prototype that automates portions of the process model and incorporates selected techniques from the Software Reengineering Study Report (Ref. 2). Using the RET, the engineer nondestructively *develops* a new Ada program by *reusing* parts of the original FORTRAN design and implementation, as opposed to *changing* the original FORTRAN into Ada.

For example, an engineer can run an automatic packaging routine that extracts FORTRAN subprograms, translates their declarations into Ada, and arranges them into packages based upon their data usage. The engineer can then rearrange the resulting Ada

3

subprograms interactively. When satisfied with the package structure, the engineer can direct the system to translate statements automatically in the bodies of the subprograms.

Most existing reengineering tools fall into one of two categories:

1. Reverse engineering and redocumentation tools (Refs. 8, 9) present different views of the structure of a program, such as control flow and data flow graphs, to aid in program understanding and manual reengineering.

2. Other tools (Refs. 10, 11) support automatic translation from one language to another or automated restructuring, such as the removal of GOTOs. These tools require little human interaction, but provide little support for design recovery or improvement.

Our approach can be described as *computer-assisted reengineering*. It provides automated reverse engineering, redocumentation, and translation of low-level program entities, and also provides a combination of user interaction and automated analysis to reorganize program statements and data into new modules.

The RET relieves the tedium associated with syntactic details (i.e., differences between the source and target programming language syntax), and allows the engineer to concentrate on more important design and implementation decisions that make the reengineered system more maintainable.

The RET design is documented in the "Software Design Document for the Avionics Software Reengineering Tool (RET) Prototype System RET-SDD-01" (Ref. 4). Section 4 summarizes the RET design.

Reengineering Tool Testing and Evaluation

During the Reengineering Tool Testing and Evaluation task (Task 4), we analyzed simulation software for the F-16 Block 40 OFP. We processed the FORTRAN source code with a commercial control flow restructuring product and analyzed it with REFINE/FORTRAN. We analyzed all but one subsystem in Block 40 separately with the RET prototype. Using the RET, we constructed an Ada package structure, created dataflow diagrams, and generated Ada source code.

The testing and evaluation activities that we performed under Task 4 are documented in Section 5.3. The hardcopies that we produced of the Packager and Dataflow Diagram views appear in Vol. II (Ref. 5).

Interviews, Conferences, and Papers

We conducted requirements interviews and attended workshops to collect information during the Software Reengineering Study task, and presented several papers during the Reengineering Tool Development task.

1. We held discussions with Mr. Kenneth Littlejohn (WL/AAAF-3) to discuss AS-RET requirements on 31 August 1992 in Dayton, Ohio. These discussions revealed that the ability to *change the functionality* of the system (as opposed to just restructuring it) was important to Wright Laboratory researchers.

2. We attended the third Reverse Engineering Forum sponsored by Northeastern University in Burlington, Massachusetts, 15-17 September 1992, and collected information on reengineering tools.

3. We gave a slide presentation on ASRET entitled *Avionics Software Reengineering Technologies and Process Model Development* (Ref. 12) at the Software Reengineering Workshop sponsored by the National Security Industrial Association's Software Committee and the Embedded Computing Institute, Naval Air Warfare Center, Ridgecrest, California, 12–14 January 1993. We obtained information on current reengineering research and software tools, and made contacts with researchers and practitioners in the field.

4. We conducted requirements interviews with avionics software maintainers at the Oklahoma City Air Logistics Center (OC-ALC), Tinker AFB, Oklahoma, 10 March 1993.

5. We presented a paper entitled *A Reuse Approach to Computer-Assisted Software Reengineering* (Ref. 13) at the Fourth Systems Reengineering Technology Workshop, sponsored by the Naval Surface Warfare Center, Dahlgren Division, White Oak Detachment, Monterey, California, 2-10 February 1994. The conference was billed as a workshop, but it was actually conducted as a symposium with little time for discussion.

6. We presented a paper entitled *An Interactive Reengineering Tool for Constructive Language Translation* (Ref. 14) at the Software Engineering Techniques Workshop on Software Reengineering, sponsored by Carnegie Mellon University and the Software Engineering Institute (CMU/SEI), Pittsburgh, Pennsylvania, 2-4 May 1994.

7. We wrote a paper entitled *A Reuse Approach to Software Reengineering* (Ref. 15) which is scheduled for publication in the June 1995 issue of the *Journal of Systems and Software*.

1.4 FINDINGS

During Task 1, we conducted a broad review of the state of software reengineering. An important insight that we gained during the study is that the motivations, activities, and results that characterize the state of the practice in reengineering are not homogeneous. The implication is that any successful reengineering effort must be highly targeted. We decided to focus the RET prototype development effort and restrict its scope based upon the *primary* needs of our sponsor. The RET prototype would be a language translation aid that automates as much of the job as is practical, leaving the rest to the engineer.

In Task 2, we developed a reengineering process model that specifies a sequence of tasks to reengineer a program written in FORTRAN to Ada. We developed a RET prototype that helps the engineer develop an Ada system by reusing parts of the existing FORTRAN system. We built up the RET capabilities incrementally to mitigate risk by devising transformations as needed for a sequence of FORTRAN programs, chosen to progressively introduce more and more elements of the FORTRAN language.

We are convinced after exercising the RET prototype that the ASRET process model is sound and the top-down reengineering style that it encourages is effective. One of our goals for the RET prototype was that it help the engineer capture as much of the existing design from the original program as possible, and the RET accomplishes this.

The principles embodied in the RET prototype and the techniques that it implements are extensible to other languages, but we are now aware that it is no simple matter to change the RET to translate between other source and target languages. This underscores the importance of selecting reengineering techniques appropriate to the project at hand.

We explored the F-16 OFP simulation system with the RET prototype during Task 4. We found that the RET views are of great practical value despite the technical imperfections that we grappled with in Task 3. We found that they summarized salient features of the subject system and focused our attention on key areas, while providing information that is not directly accessible from the source code.

One of the goals of the ASRET project was to research and develop a *prototype* for avionics support software reengineering. *The RET prototype has the potential for reducing the resources needed to reengineer avionics support software. It would help human engineers produce more maintainable systems if it were developed into a product.*

6

1.5   RECOMMENDATIONS

To demonstrate the value of the RET prototype, we recommend that software maintainers participate in an experiment, i.e., a Beta test, using an enhanced RET to reengineer an application in a production environment rather than in a laboratory. The RET prototype needs some improvements before production software maintainers who are not experts in language processing could achieve productivity with it. We have already identified some limitations of the RET prototype in Section 7.2, and we recommend addressing these before the Beta test. We also recommend preparing the RET for the test by improving or adding certain capabilities.

> *The next step towards inserting the reengineering technology that we have developed is to transform the RET from a laboratory prototype to a production tool that avionics software maintainers will evaluate on mission essential / critical applications.*

We will probably have to rely on experience reports to evaluate the RET. An engineer who's goals are consistent with those of the RET should be the final arbiter and must answer the question: "Would you use the tool again?" Based upon our experience with the RET prototype, we would expect a qualified affirmation. The modification of large computer programs will remain a most difficult undertaking, but an enhanced RET will increase the *value* of the end product, where value is a function of quality and cost.

The RET prototype will find a niche in reengineering. It relieves the engineer from syntactical minutia, i.e., differences between the source and target programming language syntax, that divert attention from the more important design and implementation decisions requiring human judgement. By concentrating on tasks that are well-suited to automated support, the RET prototype will reduce the resources needed to reengineer avionics support software and will help the human engineer produce a more maintainable system.

1.6   REPORT ORGANIZATION

Section 1 introduces the ASRET project goals and task structure. Section 2 summarizes the results of the Software Reengineering study that we performed in Task 1. Section 3 defines the ASRET Reengineering Process Model that we created under Task 2. Section 4 provides an overview of the RET design and the RET development activities of Task 3. Section 5 describes how we exercised the RET during Task 4 and narrates our production of several RET views. Section 6 reveals RET prototype capabilities in the context

of reengineering a sample application. Section 7 explains certain characteristics of the RET prototype implementation. Section 8 documents the development platform software and hardware. Section 9 presents our conclusions and recommendations.

Appendix A provides vendor information for nondevelopmental software included in the RET prototype. Appendix B describes recommended enhancements to the RET prototype. Appendix C defines acronyms used in this document.

The Avionics Software Reengineering Technology (ASRET) Project Final Report, Volume II, Reengineering Tool (RET) Diagrams (Ref. 5) contains hardcopies of the graphical views that we created during Task 4.

2.   SOFTWARE REENGINEERING STUDY

For Task1, Software Reengineering Study, we performed an extensive investigation of existing reengineering and reverse engineering processes, techniques, and tools. The study comprised two parts: the *Literature Survey*, and the *Software Survey*. We documented the results of the study in the Task 1 report, entitled "Avionics Software Reengineering Technology (ASRET) Software Reengineering Study Report" (Ref. 2). We briefly summarize the report below.

2.1   LITERATURE SURVEY

For the Literature Survey, we conducted a broad review of the state of software reengineering. We identified existing reengineering tools, software products, and techniques that that we thought were both relevant to ASRET and implementable. We screened and organized reengineering literature to distinguish the most promising methods consistent with the ASRET objectives.

We were most interested in discovering results that had been demonstrated and proven effective in improving software. We reviewed the literature in the areas of software quality metrics, program understanding, restructuring transformations, graphical and internal representations, and hypermedia.

The results that we presented in the Software Reengineering Study Report (Ref. 2) are expository rather than analytic as the requirements and design decisions were to be made during the subsequent tasks. We organized the literature into broad categories and summarized each area. We also provided detailed notes on the literature that we reviewed. For each article, we provided a synopsis of *those aspects* of the work that we thought were *relevant to ASRET.*

An important insight that we gained during the study is that the motivations, activities, and results that characterize the state of the practice in reengineering are not homogeneous. The field is bound by no more specific common interests within the community than is, say, engineering. This is neither a detraction nor a commendation. It is a recognition of diversity.

The implication is that any successful reengineering effort must be predicated on a particular class of problems and the work must be tailored to the specific problem domain. A reengineering solution must be directed at the root cause of the problem to be effective, and must not just seek to alleviate the symptoms.

We discovered no broad spectrum reengineering nostrum for improving software. Every method that we investigated involved tradeoffs. We resolved, then, to focus the RET prototype development effort and restrict its scope based upon the *primary* needs of our sponsor. The RET prototype would be a language translation aid that automates as much of the job as is practical, leaving the rest to the engineer.

2.2 SOFTWARE SURVEY

Process — For the Software Survey, we identified and described a list of software tools relevant to ASRET. We developed a set of screening criteria based on our understanding of WL/AAAF needs in order to reduce the size of the list. We screened the tools, and presented details on the 38 remaining tools which best fit our needs.

We classified the 38 tools into *application domains* according to their principle functions or purposes. Each tool appears in as many domains as needed to describe the tool's functionality. The application domains are:

1. Reverse Engineering
2. Implementation
3. Forward Engineering
4. Translation
5. Redocumentation
6. Restructuring
7. Reusability
8. Analysis.

We defined a set of software characteristics for each application domain to describe various aspects of the tools. We also defined software characteristics for the following three categories which apply to all tools regardless of their functionality:

1. Host Platforms
2. Maturity
3. Usability.

We defined a total of 28 distinct software characteristics and described the remaining tools in terms of those characteristics.

Results — We identified five tools for further investigation.

1. REFINE/FORTRAN
2. Software Refinery
3. SPAG
4. Software Reengineering Environment (SRE)
5. Arch.

We would eventually include REFINE/FORTRAN, Software Refinery, and SPAG in the RET environment during the Reengineering Tool Development task described below. The Navy evaluated the SRE to translate CMS-2 source code to Ada. Arch is a proprietary tool developed by Robert Schwanke of Siemens Corporate Research, Inc. We provided relatively long abstracts for both SRE and Arch in the Software Survey.

We decided against pursuing SRE because the prototype was language-specific and we would have had to tailor it for FORTRAN in order to use it in ASRET. Our experience on ASRET has vindicated this decision; we have learned while developing the RET that the most difficult aspects of the type of reengineering problem that we chose to address are intimately related to specific programming language features peculiar to the source and target languages.

Arch (Ref. 16) is a proprietary tool and, while we were very interested in its capabilities, we didn't have access to it. We did contact Dr. Schwanke at Siemens, but he told us that he could not provide additional information on Arch. Arch apparently has much in common with the Rigi tool developed by Dr. Hausi Müller.

Rigi (Ref. 17) did not make it through the screening process of the Software Survey, but we became more interested in it prior to the Reengineering Tool Development task. We contacted Dr. Müller and learned that Rigi was also a proprietary tool.

Despite our lack of access to these tools, the Packager component of the RET is inspired by the work of Schwanke and Müller, and also by Hutchens (Ref. 18). Their projects share a common thread of component clustering via similarity metrics and the authors cite much of the same research.

3.   REENGINEERING PROCESS MODEL

The reengineering process model as applied to the RET domain is illustrated in Figure 1. Steps in the process label the boxes in the figure, and inputs and outputs for each step label the icons between boxes.

The process model specifies a set of tasks (the steps of the process) that should be performed and the sequence in which they should be performed to reengineer a program written in FORTRAN to Ada. The process model also specifies the information necessary
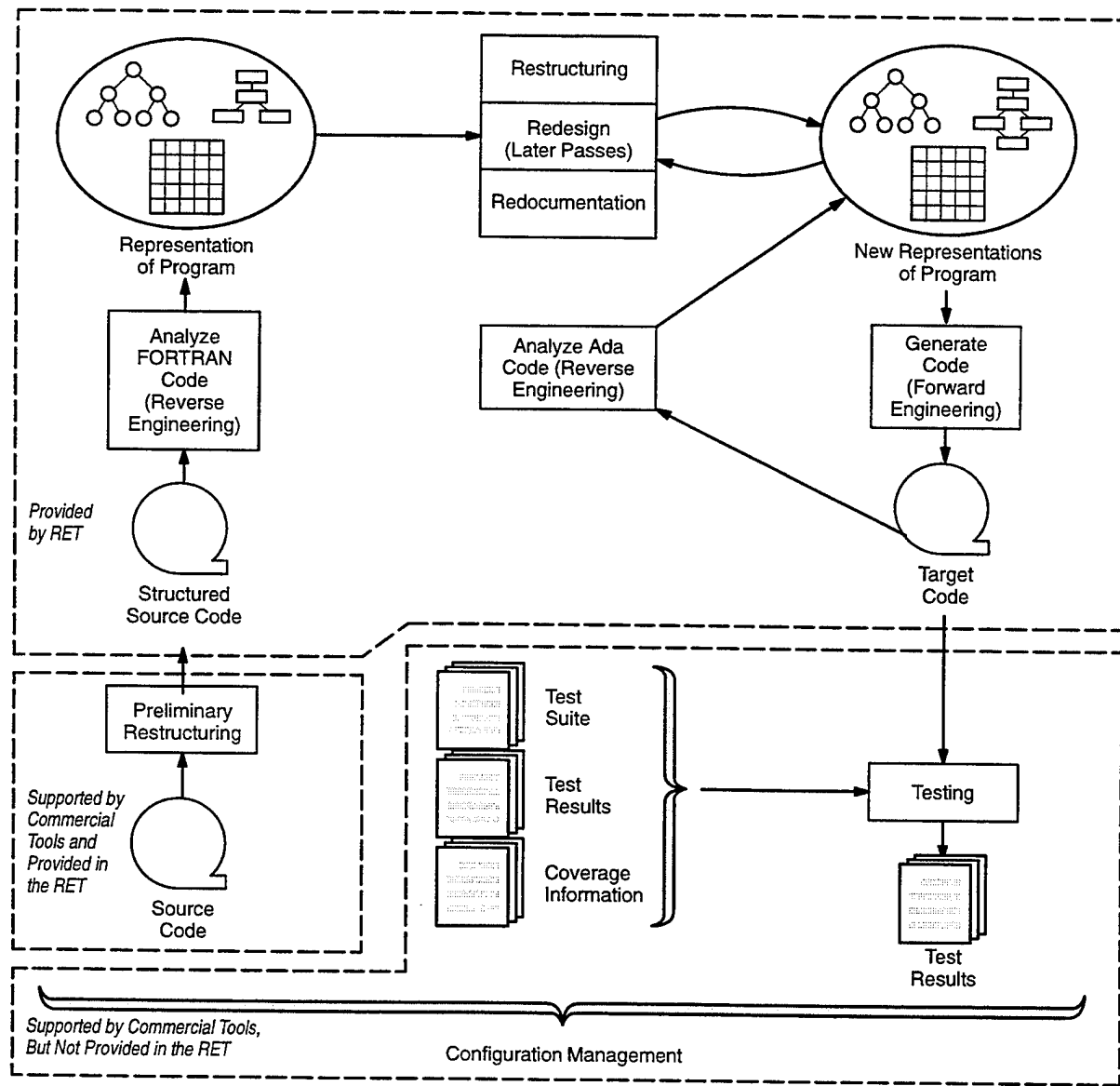

Figure 1   ASRET Reengineering Process Model

and desirable to support these tasks. The process model does not specify how the tasks are to be performed (i.e., they might be automated, as many are in the RET, or they might be performed manually).

The first step in the process model is to perform some preliminary restructuring of the source code of the original implementation. Preliminary restructuring improves the layout of the source code by removing unstructured program constructs, such as GOTO statements, dead code, and implicit types. Preliminary restructuring is separated from the later restructuring step because it can be completely automated by commercial tools, and placed first in the process model because the structured version of the source program is usually easier to analyze, understand, and restructure.

After preliminary restructuring is complete, the RET analyzes the improved source code and constructs representations of the program. Some of the representations, such as abstract syntax graphs (ASGs) and symbol tables, are machine-readable representations used only by automated restructuring and redesign tasks. Others, such as flow graphs and structure charts, aid in program understanding, redocumentation, and manual restructuring. For manual restructuring, the set of representations contains a source code listing.

The engineer performs the restructuring, redesign, and redocumentation steps multiple times, each time building upon the results of the previous pass. A multipass approach is necessary because it is easier and less error-prone to reengineer a large program in stages, verifying the program after each pass. Restructuring (i.e., changing the structure of the program without changing its functionality) is performed first, possibly in several passes. These passes perform the following steps:

- *Macro control restructuring* groups statements and control structures of the program into modules, such as procedures, functions, and packages. This includes recovering modules of the original program, generating new modules, and specifying a declaration nesting structure for modules.

- *Macro data restructuring* groups data items, such as types, variables, and constants, and associates them with modules created during macro control restructuring. This includes recovering data groupings of the original program, creating new groupings, and creating abstract data types and records.

- *Micro control restructuring* manipulates individual control structures. This includes the translation of individual statements and functionality-maintaining alterations, such as code lifting (Ref. 19).

- *Micro data restructuring* manipulates individual data items. This includes actions such as translating, changing names, changing types, creating symbolic constants, and changing the scope of variables.

13

Macro control and data restructuring should be performed first to develop a modular structure for the target system, followed by micro control and data restructuring to restructure individual components of the program.

After restructuring is complete, the RET generates code in the target language and the program is tested to ensure that the restructuring did not introduce any errors or undesired functional changes. The test data of the original program can be used and the results compared with the results of testing the original program. In many cases, the test data will need to be reengineered to work with the reengineered program.

Any differences in the results of testing indicate the introduction of an unexpected functional change during restructuring. Coverage analysis must be performed during the testing of the target code because restructuring can introduce or alter control and data characteristics of the program. When an error in the target program is indicated, the program can be corrected by amending the target code directly or by restructuring the representations and regenerating the target code.

Once the engineer has restructured the program and created a functionally equivalent program in the target language, he can perform additional restructuring and redesign actions on the program. These steps use the same set of actions (i.e., macro control, macro data, micro control, and micro data), but have different goals.

Further restructuring improves the structure of the program without changing its functionality. The goal of redesign is to change the functionality of the program (e.g., to correct design flaws or improve the design). If the engineer edited the target program code to correct errors indicated during testing, the RET analyzes the code to generate representations before performing subsequent restructuring and redesign.

The RET performs redocumentation simultaneously with the restructuring and redesign steps and can save the generated representations for documenting the program structure and design. Volume II contains samples of documentation generated by the RET prototype. Furthermore, the engineer can add comments and annotations during restructuring and redesign as he gains insights about the code or design.

The RET reengineering process model includes modern software development processes, such as continuous testing, iterative restructuring and redesign, and configuration management. The process model is a specialization of the Chikofsky-Cross process

management. The process model is a specialization of the Chikofsky-Cross process model (Refs. 7, 8). The entire Chikofsky-Cross model is represented, although there are differences:

- Program management extensions to the process model (Ref. 20) are included, such as configuration management and testing.

- Easily automated steps, such as preliminary restructuring, are separated so they can be addressed by commercial tools.

- Chikofsky-Cross steps are decomposed, such as restructuring into macro control, macro data, micro control, and micro data restructuring.

- Iteration steps that are implicit in the Chikofsky-Cross process are explicitly introduced.

4.   REENGINEERING TOOL DEVELOPMENT

This section describes Task 3, Reengineering Tool Development. Section 4.1 describes salient features of the RET prototype design. The full design is documented in the Software Design Document for the Avionics Software Reengineering Tool (RET) Prototype System RET-SDD-01 (Ref. 4). Section 4.2 records the major development activities of Task 3.

4.1   RET DESIGN OVERVIEW

We implemented the RET prototype by integrating a number of Commercial Off-The-Shelf (COTS) tools and writing transformation rules and user interface code in the environment that they provide. Table 2 lists the nondevelopmental software products that we included in the RET prototype and provides references to the associated documentation. Appendix A contains vendor information. The RET prototype architecture is described in Section 4.1.3.

- *Software Refinery* is a software development environment from Reasoning Systems that is specialized for language processing applications. It comprises three components: REFINE, INTERVISTA, and DIALECT. The components provide a programming language and database, a tool kit for building user interfaces, and a parser/printer generator, respectively.

- *REFINE* provides Software Refinery with an object-based database and a wide-spectrum specification and programming language. It includes development features for compiling, executing, and debugging REFINE programs, and utilities for browsing and manipulating the object base at a low level.

Table 2   Nondevelopmental Software

REFERENCE	PRODUCT NAME	SUPPLIER	DESCRIPTION
21	Software Refinery	Reasoning Systems	language processing environment
21	REFINE	Reasoning Systems	Software Refinery component
22	INTERVISTA	Reasoning Systems	Software Refinery component
23	DIALECT	Reasoning Systems	Software Refinery component
24	REFINE/FORTRAN	Reasoning Systems	language processing tool
25	REFINE/Ada	Reasoning Systems	language processing tool
26	plusFORT (SPAG)	Polyhedron Software	control flow restructurer
27	X-Windows	MIT X Consortium	graphical communications protocol
28	GNU Emacs	Free Software Foundation	extensible, customizable text editor

- *INTERVISTA* provides Software Refinery with basic Graphical User Interface (GUI) facilities that Reasoning Systems found to be the most convenient for developing language processing interfaces.

- *DIALECT* provides Software Refinery with a grammar specification language, a grammar parser, and parser and printer generators that create parsers and printers from grammars.

- *REFINE / FORTRAN* is an application built on top of Software Refinery by Reasoning Systems. It is used to reverse engineer and redocument FORTRAN code.

- *REFINE / Ada* is an application built on top of Software Refinery by Reasoning Systems. It is used to reverse engineer and redocument Ada code.

- *SPAG* is a component of the plusFORT product written by Polyhedron Software Ltd. The plusFORT product is a FORTRAN restructuring tool kit and the SPAG component performs control flow restructuring, among other things.

- *X-Windows* is the ubiquitous graphical communications protocol and the *de facto* standard interface for building GUI applications on Unix systems. The protocol is supported by the MIT X Consortium, but the software that implements it is provided with the hardware platform by the hardware vendor.

- *GNU Emacs* is an extensible, customizable, Lisp-based display editor provided by the Free Software Foundation.

4.1.1   The User's Perspective

The RET prototype comprises two distinct logical parts called the Left-Hand Side (LHS) and the Right-Hand Side (RHS). The LHS provides views of the original FORTRAN program, or subject system. The RHS provides views of the Ada program being developed (i.e., the target system). The LHS allows the engineer to navigate and view aspects of the subject system, but does not support changing the subject system.

The RHS supports constructing, refining, viewing, and navigating the target system. The engineer constructs a basic structure for the RHS (macro restructuring) using information extracted from the LHS. Once the basic structure of the RHS is established, the engineer refines the target system (micro restructuring) on the RHS.

Semiautomated RET prototype components support construction activities; they suggest large-scale reorganizations of the subject system and populate the RHS with the basic structure of the target system. The components that support refinement allow the engineer to apply knowledge, which is beyond the RET, and human insight, which is lacking in the semiautomated support provided by the RET, to modify and improve the RHS representations.

The RET prototype supports engineering an Ada program by reusing and transforming parts of the FORTRAN program. The process is iterative as illustrated in Figure 2.

- The engineer explores views of the original FORTRAN program that the RET generates on the LHS.
- The engineer selects LHS entities such as subroutines, statements, or data elements of the original FORTRAN program.
- The RET transforms the LHS entities and incorporates them into the RHS.
- The engineer may explore views representing the Ada program on the RHS.
- The engineer interactively or automatically refines the RHS through the views.

The engineer repeats the cycle, exploring the LHS to select additional FORTRAN entities to reuse in building the RHS. The graphical user interface presents the LHS and RHS views to the engineer while the object-based database manages the underlying data structures or internal representations.
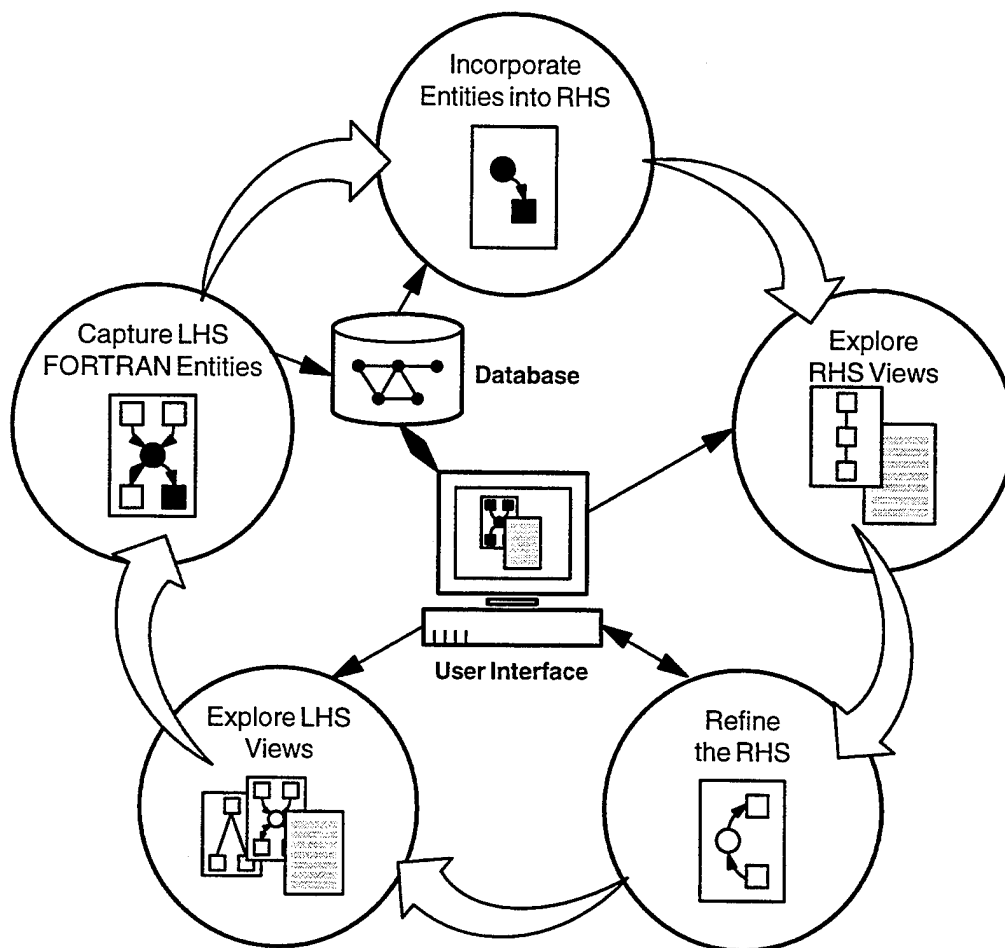
Figure 2   Developing Ada by Reusing FORTRAN

18

The RET approach to reengineering is to create a new program on the RHS, reusing components of the LHS. Structured design and programming principles are compatible with the RET and the ASRET Process Model described in Section 3. The specific steps that the engineer takes to apply the ASRET Process Model when using the RET prototype are illustrated in Figure 3.

1. The RET prototype constructs the package and subprogram structure for the RHS. It captures the subprogram structure from the LHS, transfers it to the RHS, and clusters the subprograms into Ada packages.

2. The engineer then refines the RHS structure so that related subprograms and data items are grouped together into packages.

3. The RET prototype moves data items and type declarations from the LHS to the packages and subprograms on the RHS to which they are most closely related.

4. The engineer then refines and redistributes the declarations. For example, data items that are closely related but used by several subprograms might be in different packages, and thus need to be grouped together. As another example, data items might be moved into or out of a package's private part to reflect their scope.

5. At this point, the modular structure of the program has been designed. The RET prototype transforms statements from the LHS and moves them to the bodies of the RHS subprograms and packages.

6. The engineer may then refine individual statements on the RHS to tune the RHS structure.
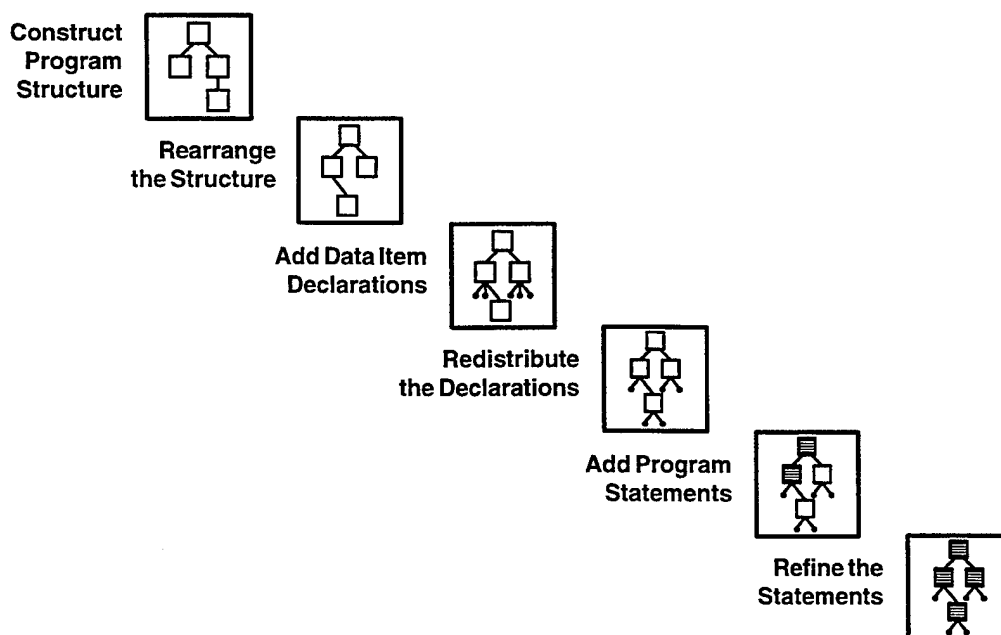
Figure 3   Incorporating Macro and Micro Entities

4.1.2   Representations

Internal Representations (IRs) are LHS and RHS data structures. The RET proto-type provides two primary internal representations (PIRs): the Abstract Syntax Graph (ASG) and the Symbol Table (ST). An ASG is a detailed representation of a program, spe-cifically, a parse tree. It is a data structure formed from objects and attributes. The classes of objects, types of attributes, and rules that define valid representations of FORTRAN (Ada) programs comprise the FORTRAN (Ada) *domain model*. The domain model is an augmented grammar that is input to DIALECT. It is a specification for the parser and printer that DIALECT generates.

The RET prototype uses the parsers and printers supplied by REFINE/FORTRAN and REFINE/Ada. We have made slight extensions to the domain models to store addi-tional analysis information produced by the RET. The RET prototype uses the REFINE/FORTRAN symbol table, also with slight extensions for analysis information.

Secondary internal representations (SIRs) are derived from the PIRs. The SIRs are the underlying data structures for the views presented to the engineer. The Software De-sign Document (Ref. 4) describes the SIRs. The RET provides the views listed in Table 3.

Table 3   Reengineering Tool (RET) Views

LHS	RHS	VIEW	NAME	DISPLAYS
✓	✓	SCL	Source Code Listing	FORTRAN or Ada source code
	✓	PACK	Packager Diagram	Ada package structure
✓		DED	Declaration Diagram	FORTRAN declaration nesting structure
✓		CD	Call Diagram	FORTRAN subprogram calling structure
	✓	DFD	Data Flow Diagram	Data flow through the Ada program

- The Source Code Listing (SCL) shows the FORTRAN source code (after proc-essing by SPAG) on the LHS and the generated Ada code on the RHS.

- The Packager view (PACK) shows the package and subprogram nesting struc-ture, and provides a graphical interface for developing the Ada package struc-ture on the RHS.

- The Declaration Diagram (DED) documents the FORTRAN system declara-tion structure. The PACK provides similar information for the Ada system, so we did not create an Ada version of the DED.

20

- The Call Diagram (CD) documents the FORTRAN calling structure. The PACK provides call diagrams for the Ada system, so we did not create an Ada version of the CD.

- The Dataflow Diagram (DFD) documents the Ada system data flow. The PACK provides some data flow information that is derived from the FORTRAN system, so we didn't create a version of the DFD for the FORTRAN system.

Section 4.2.2 describes the RET prototype views in greater detail and recounts our experiences in developing the views.

4.1.3 Architecture

Figure 4 shows the RET architecture. It depicts the organization of major RET components, and indicates the data flow relationships among them. The Preliminary Restructurer (PR) performs control flow restructuring. The Source Code Processor (SCP) generates the LHS PIRs.

The engineer constructs the RHS PIRs using the Packager (PACK) and Transformer (TRAN). The Representation Generator (RG) creates the SIRs on both sides from the PIRs. The User Interface and Display (UID) creates the corresponding views, and provides the means by which the engineer interacts with the views on both sides and alters the views on the RHS. The Transformer implements the changes by transforming the RHS PIRs, and the Representation Generator propagates the changes to the RHS SIRs. The User Interface and Display refreshes the RHS views in response to the changes.

The File System Interface (FSI) manages external persistent data and the Object Base (OB) manages internal data in the form of data objects, attributes on these objects, and relationships between objects.

The views, PIRs, and SIRs are thus interdependent, but the engineer need not be aware that the PIRs and SIRs exist. Any changes that the engineer makes to the target system through the views provided by the User Interface and Display appear to affect the views exclusively. The components are described in more detail below.

Preliminary Restructurer — The Preliminary Restructurer (PR) restructures the control flow of the original FORTRAN source code by eliminating branches into or out of loops and decisions. It eliminates all GOTO statements, leaving only the structured programming constructs: sequence, selection, and iteration. We refer to this specialized form of restructuring as control flow restructuring to distinguish it from the more general concept of restructuring described in Section 2.
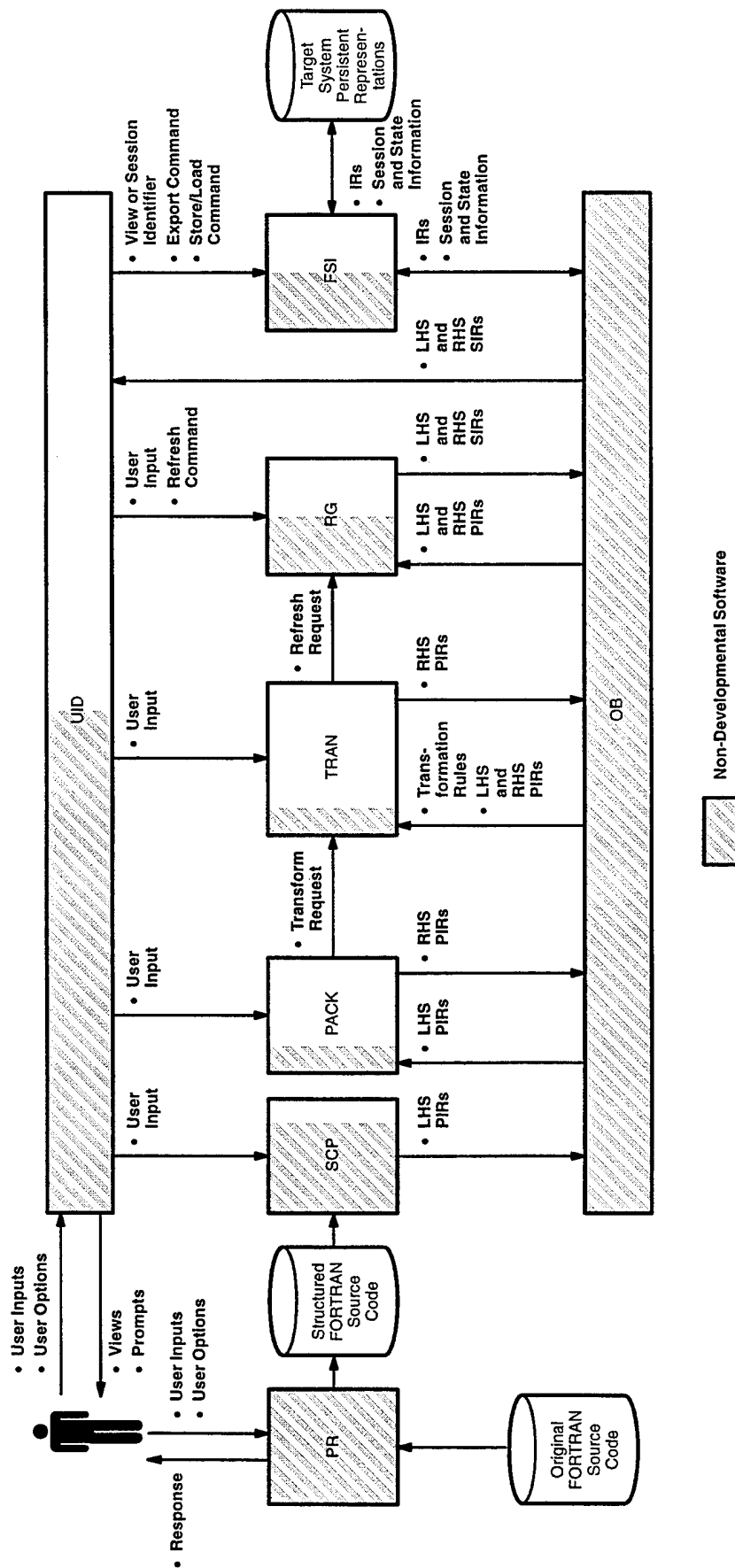
21

Figure 4 RET Architecture

Non-Developmental Software

22

The Preliminary Restructurer is applied as a pre-processing step and the RET assumes that the FORTRAN source code has already undergone control flow restructuring. There are two reasons for this design. First, the subject system is not always poorly structured with respect to control flow, so the step should be optional. Second, the design allows any control flow restructuring product to be used without integrating it into the RET.

Source Code Processor — The Source Code Processor (SCP) reads the FORTRAN source code, performs analyses, and generates the PIRs. The PIRs represent the structure of the program, semantic information about program components, and data flow information.

Representation Generator — The Representation Generator (RG) creates the LHS and RHS SIRs. The Representation Generator generates each SIR when the engineer requests its corresponding view. Once the RHS SIRs are created, they may become inconsistent with the PIRs as the latter are restructured. The engineer can request a refresh, which regenerates the SIRs from the current PIRs.

The Representation Generator creates SIRs for the DFD, CD, and DED views. The Representation Generator produces the DFD according to a method for creating Hierarchical Data Flow Diagrams given in (Ref. 29). It generates the CD in a straightforward manner from information in the ST. The DED is a canonical organization of information in the ST.

The Representation Generator does not create SIRs for the SRC views. The SRC view is produced by the DIALECT printer (Refs. 23, 30).

Restructurer — The Restructurer (RES) component comprises the Packager (PACK) and the Transformer (TRAN). The Restructurer helps the engineer develop an Ada program on the RHS by transforming and reusing components of the original FORTRAN program from the LHS. The Packager assists the engineer with macro restructuring. The Transformer supports both macro and micro restructuring.

The restructuring activities that the engineer performs with the Restructurer manipulate the PIRs exclusively, by initially creating RHS symbol tables and ASGs and then by populating and transforming them. The engineer enters the information through the views and, once the RET transforms the PIRs, it regenerates the SIRs and refreshes the views. Thus, the underlying PIRs and SIRs are hidden from the engineer and it appears as if the RET is directly transforming the views.

Packager — The Packager constructs or initializes the RHS ASG. It recognizes subprogram, object, and type entities from the LHS PIR and requests the Transformer to transform them from the LHS domain model to the RHS domain model, and to insert them into the RHS ASG. (Each domain model (Ref. 30) is an object-based database schema specifying the objects and relationships between objects necessary to represent LHS or RHS information.) The Packager builds an Ada ASG on the RHS and uses REFINE/Ada for semantic analysis.

The Packager groups the entities into Ada packages on the RHS by applying interactive clustering techniques based upon algorithms developed by several researchers (Refs. 16, 17, and 18). The clustering techniques provide a first approximation to a reasonable Ada package structure. Usually, the engineer needs to interactively refine the generated grouping.

Transformer — The Transformer assists the Packager with both macro and micro restructuring. For macro restructuring, the Transformer automatically transforms low-level entities from the FORTRAN domain model to the Ada domain model. It transforms them by applying rules that insert subgraphs and other information into the Ada PIRs corresponding to subgraphs and information in the FORTRAN PIRs.

For micro restructuring, the Transformer implements changes to low-level Ada entities by allowing the engineer to select a portion of the Ada program under development and change it by applying a rule or by editing, deleting, or inserting text.

User Interface and Display, File System Interface, Object Base — The RET provides two external interfaces. The engineer communicates with the RET through the User Interface and Display (UID). The UID shows the views, prompts the engineer for input, receives commands and selections from the engineer, and delivers the commands and selections to the other components.

The File System Interface (FSI) is responsible for the storage and retrieval of persistent data. The FSI inputs the (FORTRAN) source code of the subject system, reads and writes intermediate data, outputs the (Ada) source code for the target system, and outputs other subject and target system views. The intermediate data is stored in the Object Base.

4.2   DEVELOPMENT ACTIVITIES

This section relates some of our experiences in developing certain features of the RET prototype listed in Table 4. Although we started with a specific design (Ref. 4) for the RET prototype, we identified important enhancements as we worked with the software and analyzed the Fire Control Radar (FCR) simulation code for the F-16 Block 40 OFP.

4.2.1   Features

We worked exclusively with the FCR subsystem during the Reengineering Tool Development task, developing features that we needed to reengineer the FCR code. In this section, we discuss the development of these features, listed in Table 4. We elaborate on some of them in Section 6 within the context of using the RET prototype to reengineer the FCR code.

Table 4   RET Prototype Features

(1)	Enhance copy-term	(10)	Parentheses in Source Code Views
(2)	Source Code Views	(11)	Saving Views
(3)	LHS/RHS Pointers	(12)	Hardcopy Output
(4)	SPAG Component of plusFORT	(13)	Ada Code Generation
(5)	Transformations	(14)	Data Flow Diagram
(6)	Packager Algorithm	(15)	Intrinsic and External Subprograms
(7)	Packager Editing	(16)	Constant Packages
(8)	Type Deduction	(17)	Global Variable Distribution
(9)	Type Conversion		

(1)   Enhance copy-term — We used a REFINE utility called *copy-term* to copy FORTRAN ASGs to the RHS before transforming them to avoid destroying the original FORTRAN ASGs. Copy-term only copies attributes that define the tree structure of the ASG, but some information is stored on non-tree attributes. We added a function to the REFINE hook that copies the other attributes that the RET prototype needs.

(2)   Source Code Listings — The Source Code Listing (SCL) views were among the first views that we integrated. The RET prototype calls upon features of REFINE/FORTRAN and REFINE/Ada to display the source code, so the only code that we had to implement was the so-called "glue" code to integrate them.

(3)   LHS/RHS Pointers — The RET prototype maintains pointers between the LHS and RHS representations. The RET prototype initially creates the pointers when it builds the RHS ASG. It reconstructs the pointers when it regenerates and parses Ada code on the RHS. The pointers are used in hyperlinking.

(4) **SPAG Component of plusFORT** — We installed the plusFORT product, including the SPAG component. SPAG is the control flow restructurer that implements the Preliminary Restructurer (PR) component of the RET. We processed all of the FCR code with SPAG.

(5) **Transformations** — We first developed the simple transformations such as those for the assignment statement. Later, we added the more complicated transformations such as those for expressions and intrinsic functions. Lastly, we added the capability to move comments from the LHS to the RHS.

(6) **Packager Algorithm** — As we were developing the algorithm for the Packager component, we realized that an interactive view would be very helpful. We therefore implemented a graphical interface for the Packager. As we worked with the Packager, we decided that it contained all of the information necessary for a RHS Declaration Diagram (DED) and that the graphical view was better than the text-based LHS DED. For these reasons, we didn't implement a separate DED on the RHS.

(7) **Packager Editing** — After experimenting with the Packager, we decided that an engineer would need to edit the module groupings that it creates. We then added the editing features to the Packager. As we exercised the Packager, we added additional menu items and features that we felt were needed. Examples are the commands for selecting several nodes in order to move them to another graph and features for arranging the nodes and edges of the graph.

(8) **Type Deduction** — We implemented a feature to deduce bit-field types for the RHS. These are needed because of differences in the FORTRAN and Ada type systems. The details of the problem and its resolution are described in Section 7.1.

(9) **Type Conversion** — We generalized the support for Ada implicit and explicit type conversions. The conversion techniques that we initially implemented only converted to or from bit field types. The current implementation supports conversion between other types, such as integer and real.

(10) **Parentheses in Source Code Views** — REFINE/FORTRAN stores parentheses along with the comments because they aren't part of the ASG. We built menu options in the Ada Source Code view to add and remove balanced pairs of parentheses and we enhanced the Transformer to recognize the parentheses in the comments and transform them into Ada.

(11) Saving Views — We added support to save the Packager and Data Flow Diagram views to disk and reload them in subsequent REFINE sessions, i.e., across logins. We implemented this through the Persistent Object Base (POB) YOYO* system. The approach that we took was to save the PACK and DFD views separately, as opposed to saving the state of an entire session. We did not implement the capability to save an entire RET prototype session because it wasn't necessary when we implemented persistent storage.

At the time that we implemented persistent storage, the RET prototype did not perform much initialization processing. When we moved processing for other features to the initialization phase to improve on-demand performance for those features, we increased the initialization processing requirements. There is now enough initialization processing to warrant saving an entire session, but we didn't implement that enhancement. (See Section 9.2.1, *Save intermediate analysis data to disk*.)

One implementation for saving the session would be to attach the Secondary Internal Representations (SIRs) to the REFINE/FORTRAN and REFINE/Ada Primary Internal Representations (PIRs) and save the PIRs and SIRs to the POB. This implementation would require rewriting all of the information that REFINE/FORTRAN already saves in a .analysis file, (Section 5.3.2) including the ASGs, but would obviate reanalyzing the PIRs and speed up the RET prototype initialization.

(12) Hardcopy Output — We implemented hardcopy output by calling the Mouse Sensitive Printing (MSP) and DIAGRAM window printing functions in the REFINE YOYO hardcopy system. We initially thought that we would need to install a 4.0 Beta version of REFINE, but we were able to use the unsupported YOYO hardcopy system in the version that we were already running. The RET provides the capability to print any view by selecting the "print" item from the pull-down menu for the window.

(13) Ada Code Generation — We implemented Ada code generation by calling the REFINE/Ada printer, analyzing the resulting code to create an Ada ASG on the RHS, and reestablishing the pointers with the LHS. We encountered many parsing errors initially, but worked through them until REFINE/Ada could parse all of the code that the RET prototype generated. (We developed this feature by analyzing and generating code for the FCR.)

---

* YOYO is an acronym for You're On Your Own. The YOYO systems are unsupported releases of features that Reasoning Systems typically incorporates into the supported products in subsequent versions. The acronym derives from the fact that Reasoning Systems does not *formally* support the features (although they did help us with some of the YOYO systems when we asked).

Paradoxically, the REFINE/Ada printer can generate Ada code that the REFINE/Ada parser rejects. This is possible because the printer simply prints the ASGs that the Transformer creates, and it is possible to construct an ASG which is syntactically invalid. REFINE/Ada is very good at checking data type and scope violations.

(14) Data Flow Diagram — The RET prototype generates the Data Flow Diagram (DFD) from the RHS ASG created by REFINE/Ada. This is not the ASG that the Transformer creates. We use the ASG created by REFINE/Ada because it contains the semantic information produced by the REFINE/Ada linker. We enhanced the DFD during the development task as described in Section 4.2.2.

(15) Intrinsic and External Subprograms — The RET prototype translates an implicit FORTRAN function that does not have an equivalent Ada primitive into an Ada function call. The RET gives the Ada function the same name as the FORTRAN function that it is derived from. The RET generates an Ada package named "implicit_fns" that contains stubs for each Ada function translated from an implicit FORTRAN function. (The engineer can change this package name.)

The engineer must provide bodies for the Ada functions by, for example, supplying an existing Ada library that implements the required functions or coding the functions "by hand." The RET prototype similarly translates external subprograms, i.e., subprograms that are referenced, but not declared in the FORTRAN system.

(16) Constant Packages — The RET prototype generates Ada packages for constants, derived from the FORTRAN *PARAMETER* statements in included files, and automatically generates context clauses for the constant packages. The engineer may view the packages from the Packager view.

(17) Global Variable Distribution — We implemented a global variable distribution feature that identifies common blocks that should not be split up, i.e., memory-mapped common blocks. The RET prototype recognizes common blocks that are declared in several places or whose variables are used in *EQUIVALENCE* statements anywhere in the program. Both occurrences indicate the possibility that the program is relying on the common block residing in a contiguous block of memory.

When the RET prototype distributes the global variables, it pops up a window that shows the common blocks and highlights those that it identifies as not being memory-mapped. The engineer either accepts these or selects other common blocks. The RET prototype then distributes the variables in the highlighted common blocks.

28

4.2.2 Views

SCL — The Source Code Listing (SCL) is a textual view that shows the FORTRAN or generated Ada source code. The LHS SCL displays the FORTRAN code as it was formatted when input to the RET prototype, after it was processed by the SCP. The SCP alters the source code, so the output format is generally different from that of the original unprocessed FORTRAN code. The SCL shows individual FORTRAN and Ada subprograms, and Ada packages.

The RHS SCL displays the Ada code as formatted by the REFINE/Ada (RA) printer. When the RET prototype generates an ASG, it does *not* insert formatting information (specifically, surface syntax that would control the appearance of the code) into the ASG; the RET allows the RA printer to compute the surface syntax when it creates the Ada code from the ASG. The RET prototype subsequently calls RA to parse and analyze the Ada code.

RA creates another ASG while parsing the source code, and it contains more information than the one that the RET generated. In particular, RA annotates the ASG with sequences of characters, called surface syntax, that comprise the code. In other words, the ASG created by RA contains (in an encoded form) an exact copy of the source code that it represents. If something changes the Ada source code between the time the RA printer generates it and the time RA parses it, the new ASG captures the revised code as surface syntax.

The RET prototype uses the RA printer to create Ada code from an ASG that may contain surface syntax. The RA printer doesn't compute the arrangement of the Ada code that it creates from an ASG that includes surface syntax; it simply references the surface syntax to print out an exact copy of the source code read by the parser. An external agent, e.g., a code formatter, may thus change the appearance or arrangement of the Ada code between code generation and parsing.

The current RET prototype does not provide an automatic code formatter. The engineer may edit the generated Ada code before RA parses it, however, and the ASG and surface syntax created by the RA parser preserves any changes that the engineer makes, including those that affect the arrangement of the code.

PACK — The Packager view is a hierarchy of graphs. The hierarchy corresponds to the nesting structure of the target system. There is one graph for the Ada library, and one for each potential package. The nodes in each graph represent modules, i.e., subprograms or packages, and the edges represent either data relationships or subprogram calls. Section 6.1.1 describes the Packager view and how it is used to cluster a sample application.

The Packager view contains two kinds of edges. Thin, undirected edges depict the data sharing relationships between two nodes. A thin edge between two subprograms indicates that the two subprograms share data. A thin edge between a package, P, and a package or subprogram node indicates that at least one subprogram in P shares data with the modules that the other node represents. The edge labels list the data objects or show the subprograms in the package that share data.

A thick edge drawn as an arrow directed from one subprogram to another represents a subprogram call in the direction of the arrow. If the edge is drawn between a package and another node, then the edge may represent multiple subprogram calls and its label shows either the number of subprogram calls or the names of the called subprograms.

DED — The Declaration Diagram is a textual view that shows the declaration structure of the subject system. For the FORTRAN system, the (LHS) DED lists the subprograms and common blocks. For each subprogram, it lists the formal parameters, local constants and variables, and included files. For each variable, constant, and parameter, the DED shows the data type and describes where the data object is declared and referenced.

We had anticipated creating an RHS DED for the Ada system, but found that the Packager view served the purpose well. We referenced the DED while creating an Ada package structure with the Packager mainly to locate and view the source code for a given subprogram. After adding the capability to view source code directly from the Packager view, we found that we no longer referenced the DED.

CD — The Call Diagram (CD) is a textual view that shows the calling structure of the FORTRAN system. The CD lists the subprograms that comprise the original system and shows the subprograms that each one calls, and the subprograms that are called by each one. The CD initially shows only a list of subprograms and allows the engineer to view the additional calling information for selected subprograms.

DFD — The Hierarchical Data Flow Diagram (HDFD) (Ref. 29) is a representation specifically designed to be created via reverse engineering. It provides dataflow information recommended by the software development (forward engineering) methodology known as Transform Analysis.

The ASRET Data Flow Diagram (DFD) is based on the HDFD, but the DFD is an interactive view rather than a static diagram. We describe other differences below. The RET prototype automatically generates the DFD via the algorithm specified in Ref. 29 from the Ada ASG.

30

The DFD view is a hierarchy of dataflow graphs. The hierarchy corresponds to the calling structure of the target system. There is one graph for each subprogram *declared in the target system* that is called. This means that there are no graphs associated with undefined external subprograms or intrinsic functions (because we are not reengineering their source code).

The DFD contains two kinds of nodes. *Transform nodes* model programs and *repository nodes* model data. Three kinds of transform nodes are borrowed from the HDFD.

1. *Nonterminal nodes* represent subprograms that are associated with graphs because they call other subprograms

2. *Terminal nodes* model subprograms that call no other subprograms and thus have no graphs

3. *Body nodes* represent subprogram bodies.

Three flavors of repository nodes model data. The first two are borrowed from the HDFD and the third is a new kind that we created for the DFD.

1. *Buffer nodes* represent data objects (local variables, global variables such as those declared in Ada packages, and subprogram arguments)

2. *Data store nodes* represent external storage such as files

3. *Repository collection nodes* combine sets of repository nodes.

Arrows between the transform and repository nodes indicate the flow of data. Transform nodes are labeled with subprogram names. Repository nodes are labeled with data item names. Repository collection nodes representing more than a few (the engineer specifies the threshold) data items display the total number of data items, and the engineer may click on the collection nodes to view the individual data item names. The collections may be nested.

We did not implement data stores because the subject system contained very few *READ* or *WRITE* statements. We simplified the Data Flow Diagram view by combining repository nodes, eliminating redundant transform nodes, and eliminating nodes for intrinsic functions and external subprograms. This reduced clutter in many of the diagrams, but we found that a few of the diagrams were still hopelessly complicated, reflecting the design of the associated modules.

31

We added a capability to automatically eliminate nodes for intrinsic functions and external procedures. Intrinsic functions are those listed in Appendix D.3 of the VAX FORTRAN Language Reference Manual (LRM) (Ref. 31). Examples are SIN and SQRT. External procedures are system subroutines, listed in Appendix D.4 of the LRM, that are called from the FCR code. Examples are DATE and EXIT.

We added the capability to automatically eliminate redundant transform nodes. Benedusi defined the Hierarchical Data Flow Diagram (HDFD) (Ref. 29) to include distinct nodes representing multiple calls to a given subroutine. For example, if module A called module B three times, then the diagram for module A would contain three distinct nodes, each labeled "B," representing the three calls to module B.

The three nodes would only be connected to different nodes in the diagram if different actual parameters were used in each of the calls. For may graphs in the FCR code, parameters are not passed, so the three "B" nodes are each connected to the same set of nodes, i.e., two of the nodes are redundant. The FCR code happens to have many such redundant nodes, so we added the capability to eliminate them. The RET only generates one "B" node for the example cited.

We added the capability to combine sets of repository nodes into collections. This is analogous to combining several variables into a record structure. The strategy reduces the number of repository nodes in a graph. It is most effective if each of the repository nodes in a collection are referenced more or less by the same set of modules. We added this capability to explore its effectiveness in abstracting data usage relationships in an otherwise complicated diagram.

We did not add a user interface for specifying the groupings because the capability is experimental. The graphical results seem to indicate that the strategy is helpful if the engineer can identify related variables. It is effective, but labor intensive because the engineer must identify groups of repositories based on information not available in the DFD. We chose to group variables with similar names, e.g., CURSORX and CURSORY.

5.   TOOL TESTING AND EVALUATION

5.1   EVALUATION SUMMARY

During the Reengineering Tool Testing and Evaluation task, Task 4, we exercised the RET prototype by analyzing the simulation software for all but one of the subsystems from Block 40 of the F-16 OFP. For each one we:

1. Processed the FORTRAN source code with SPAG to restructure the control flow and eliminate certain unstructured constructs such as arithmetic IF statements and implied declarations

2. Executed REFINE/FORTRAN to analyze the restructured FORTRAN source code and saved the FORTRAN ASG and symbol tables for later use by the RET prototype

3. Fed the analysis information created by REFINE/FORTRAN to the RET prototype, which analyzed the FORTRAN source code indirectly by processing the ASG and symbol table

4. Developed an Ada package structure via the Packager component of the RET prototype and generated hardcopy versions of the Packager views

5. Generated Ada source code corresponding to the FORTRAN subprograms and organized according to the prescribed package structure

6. Generated dataflow diagrams corresponding to the Ada code and arranged the transform and repository nodes to clarify the graphs.

Volume II (Ref. 5) contains hardcopies of the Packager and Dataflow Diagrams that we created during Task 4.

5.2   SUBJECT SYSTEM

The subject system that we analyzed in the Reengineering Tool Testing and Evaluation task is the F-16 Block 40 OFP simulation software. It comprises the seven subsystems shown in Table 5 and is written in VAX FORTRAN.

Table 5 F-16 Block 40 OFP Subsystem Sizes

SUBSYSTEM	LOC*
FCR	16,202
GPS	3,818
INS	9,637
MFD	16,291
RLT	1,455
UFC	14,345
SMS	21,554
Total	83,302

*The figures for the lines of code (LOC) in this table are those reported by the unix *wc* utility and include blank lines and comments.

5.3   ANALYSIS PROCESS

5.3.1   SPAG Source Code Preprocessing

We employed SPAG as the Preliminary Restructurer component of the RET prototype. We ran SPAG on every source code file for each of the subsystems shown in Table 5. We stored all output, including the SPAG log files, on disk as documented in Section 8.2.

SPAG doesn't remove all of the GOTO statements, but it gets most of them. SPAG reformats the FORTRAN source code. It indents and inserts spaces as it deems appropriate. Overall, it does a good job, but there is at least one case where the formatting is inconsistent with our preferences.

SPAG tries to format expressions such that the white space indicates the implicit relative binding strength or precedence of the operators (which could be made explicit by parentheses). It never provides white space around multiplication (*), division (/), or exponentiation (**) arithmetic operators, but it can surround the addition (+) and subtraction (-) operators with spaces.

The formatting goal of SPAG is understandable, but it does not reproduce the existing FCR code formatting in which every operator is surrounded with spaces. Whether this formatting strategy is unfortunate or not depends on the engineer's subjective preferences or the organization's coding standards.

5.3.2   REFINE/FORTRAN Analysis

We used REFINE/FORTRAN (RFT) to analyze the subject system source code files and save the analysis results in (.analysis) files on disk. We executed RFT through its own

user interface, i.e., not invoking an RFT analysis function from the RET prototype, but invoking RFT from the shell prompt. We had developed a RET prototype interface to RFT which can be used to call the RFT parsing functions, but we found it more convenient to execute it as a separate step.

This part of the analysis is essentially a batch process and doesn't involve the RET prototype. The engineer may execute the RFT analysis functions through a shell script and completely bypass the RFT user interface. The RET prototype doesn't require the use of any RFT GUI capabilities, it only needs the .analysis file created by the RFT analysis functions.

RFT had only one problem parsing the FCR code. RFT requires spaces around arithmetic operators and there were two instances of operators in the FCR code that were adjacent to their respective operands without intervening spaces. We simply edited the source code "by hand" to add spaces where necessary.

We analyzed the subsystems shown in Table 5 separately because the RFT documentation (Ref. 23) states that the tool runs faster this way. We produced separate RFT .analysis files for each subsystem. Each of the subsystems referenced many of the same include files, for example to define parameters and common blocks.

5.3.3   RET Prototype Analysis

We analyzed each subsystem via the RET prototype. This entails loading the .analysis file created by RFT and performing some initial processing. The engineer simply specifies the .analysis file name and the RET prototype performs the analysis automatically.

The RET prototype performs analyses during the initialization phase beyond those performed by RFT and saved in the RFT .analysis file. The RET prototype initialization includes:

- Creating tables of program units and common blocks
- Creating symbol cross references for program units and common blocks
- Analyzing the surface syntax (embedded source text) in each file.

The analysis averages about one hour of wall clock time per subsystem. Section 4.2 describes the issues surrounding this aspect of RET prototype performance with respect to saving the REFINE session. It is likely that we could reduce the time required for the analysis by tuning the RET prototype code, although we did not tune the prototype.

After performing the initial analysis for a subsystem via the RET, we created the Packager and DFD views for that subsystem and generated hardcopies (Ref. 5). Sections 5.3.4 through 5.3.6 describe our efforts to create the Packager and DFD views and generate Ada source code.

We referenced the SCL, DED, and CD views that the RET prototype created for each subsystem when we formed the Packager views. While we found the DED and CD helpful during this process, we did not create hardcopies of them because they are very straightforward textual views and not particularly interesting in themselves. The SCL views are identical to the source code which we retained as per Section 8.2.

We inspected the Packager and DFD views that we created and compared portions of the views with the FORTRAN source code. We limited our inspections to portions of the views as opposed to systematically examining every element in every view because of their size. We discuss in Vol. II (Ref. 5) specific elements of the views that we created.

5.3.4   Creating Packager Views

The Packager presents the engineer with a graphical view of the subprograms of the FORTRAN system and provides an interactive mechanism for clustering the subprograms into Ada packages. The Packager automatically distributes global data items among the subprograms and packages, and allows the engineer to alter the distribution. Section 6 describes in detail the Packager view and narrates an example Packaging session.

The clustering algorithms (Section 6.1.2) implemented in the RET prototype iterate over $O(n^3)$ matrix operations where n is the *current* number of modules in a graph. (The number of modules decreases after each iteration as the clusters form.) The REFINE implementation of the algorithm executed a single iteration in about 10 minutes of wall clock time for 100 nodes. We reimplemented the algorithms in Lisp, adding declarations to avoid "consing," and an iteration now executes in about 15 seconds of wall clock time, for a roughly fortyfold improvement in processing speed.

While 15 seconds is not a short response time for an interactive system, the engineer nominally spends most of the time studying the Packager view, only intermittently initiating clustering iterations. Amortizing the computation time over the period during which the engineer interacts with the Packager, the delays did not seem at all unreasonable to us, psychologically or practically.

The Packager pops up an SCL view or Emacs buffer containing the source code of the program unit associated with any Packager node when it is requested by the engineer. If the node represents a subprogram that is not a subunit, then the associated program unit is the entire package containing the subprogram. In this case, the engineer must scroll the view to locate the desired subprogram.

The Packager allows the engineer to specify whether each module will generate an Ada subunit. The RET prototype generates a separate file containing the Ada code for each module flagged as a subunit, as required by the Ada language. We chose to specify all subprograms as subunits to facilitate browsing the source code of individual modules.

It is possible to change the RET prototype so that it automatically scrolls the SCL view or Emacs buffer to display the appropriate subprogram, but we found it convenient to simply flag all subprograms as subunits while developing the target system. We recognize that the engineer will generally want only some of the subprograms to be subunits. This is not a problem because the RET prototype can regenerate the Ada code at any time with the desired subset of modules marked as subunits.

One of our goals was to capture as much of the existing design from the original program as possible. The software that the RET prototype targets has a structured design; the program is divided into subprograms and files representing particular software requirements. Reorganizing this program under a radically different design methodology, such as object-oriented design, where subprograms are organized around data, might encumber maintainers who are already familiar with the existing application domain model.

We therefore decided to maintain the subprogram division of the original design and use Ada features that improve on it, e.g., packages. Even though we were not trying to produce an object-oriented design for the target system, we thought that clustering based upon patterns of data usage would help us discover cohesive package groupings with few interfaces that encapsulate much of the data.

Most of the subsystems contained sets of subprograms that were amenable to clustering. Many, however, contained large numbers of subprograms that formed very highly connected, sometimes nearly complete subgraphs when viewed with the data binding edges of the Packager. These portions required more intensive "manual" evaluation, i.e., source code perusal, inspection of the calling relationships, and examination of the specific data objects involved in the bindings. We found it difficult to cluster these portions of the subsystems.

Part of the reason for the resistance to clustering may be due to the fact that we did not undertake to split functions. We encountered some large sets of subprograms that were so tightly coupled that we discerned no clear component structure. Additional research might seek to determine whether splitting subprograms in these situations would lead to a larger number of cohesive clusters with less coupling.

Our recourse during the RET prototype evaluation was to take advantage of the calling information that the Packager view provides. When we encountered a refractory set of subprograms for which the Packager view of the data bindings formed almost a complete graph, we adjusted the Packager view to display the calling relationships instead of the data bindings to help identify a package structure.

The automated clustering algorithms helped identify package structures for much, but not all of the subject system. While the algorithms suggest alternative groupings and provide recommendations, the engineer must exercise judgement in analyzing the subject system. We felt disadvantaged in that we did not have much experience in the particular application and we are not involved with this application domain on a day-to-day basis. For this reason, we highly recommend a "usability testing" experiment by maintainers familiar with the target system that is the subject of the RET experiment.

The Packager provides information to the engineer. We expect that an engineer who is committed to understanding the application, or who has experience with the application will be in a better position to take advantage of the information. We did not have time during the evaluation to study the application itself as we were more focused on the RET prototype and the reengineering process. This is one of the reasons that we recommend in Section 9.3 that regular application maintainers participate in testing an enhanced version of the RET outside of the laboratory environment.

Additional research on the RET prototype might seek to identify how sensitive its effectiveness is to the engineer's experience and knowledge of the application. One problem in reengineering is the difficulty of recruiting and retaining talented engineers to work on an old application. This results in an overall decrease in the level of the maintainers' familiarity with it. The RET prototype may demand some level of application domain knowledge and experience with the subject system, but we have no experimental data by which to define metrics to quantify or bound the critical mass for the knowledge base.

5.3.5 Generate DFD Views

We generated DFDs for the FCR subsystem (Section 5.2) first. While we were in the process of creating the DFDs for the FCR, we implemented the modifications discussed in Section 4.2 for simplifying the diagrams. We spent a lot of time arranging the diagrams by dragging nodes with the mouse and we think that an automatic graph placement feature is absolutely necessary for a production quality RET in a software maintenance environment.

The ability to collect repository nodes into groups helped simplify some of the diagrams, but others remained incomprehensible. We think that application domain knowledge and a better user interface for grouping repositories may allow the engineer to further reduce the amount of detail, allowing additional data flow relationships to emerge. We expect that some data flow relationships will continue to be obscured by the complexity of the most complicated DFDs, however.

It would be interesting to investigate the potential benefits of filtering repositories in the DFDs. Individual diagrams, or the entire set of diagrams in a DFD could be filtered to show specific kinds of repositories. For example, the DFD could show only repositories representing global variables, or those for a specific set of variables of interest to the engineer. The engineer would be able to focus in on data areas of interest, formulating specific queries and regenerating the DFDs in response. Research on filtering criteria or classes could identify promising domains to improve the maintainer's understanding during a reengineering session.

An automatic graph placement capability is essential for this feature to be practical. One difficulty that the system would need to address is the possibility that transform nodes might appear in different locations from one generation to the next. The tradeoff is that fixing the transform nodes' positions to maintain the engineer's frame of reference constrains the graph placement algorithm's ability to optimize the arrangement of nodes in the graph.

Some parts of the diagrams remained cluttered after we implemented several simplifications to the DFD. These diagrams point out areas that require additional "manual" analysis by the engineer. In these cases, the RET prototype identifies areas that may demand more attention from humans during the reengineering process.

We produced high-level graphs for each subsystem that show the overall flow of data among major parts of the subsystem. When arranging a newly-generated graph, we would first try to position the nodes so that no edges crossed. Many of the graphs were planar, or very nearly so. The notable exceptions were the highly connected graphs with many nodes.

We realize in retrospect that disallowing editing of the generated DFDs was a mistake. An engineer might find it useful, for example, to combine nodes or edit the node labels. The repository collections combine nodes, but the groups must be specified in advance. The Hypertext Annotation (HA) feature that we deferred (Section 7.2) would have allowed the engineer to add textual commentary, but DFD graph editing capabilities would be welcome.

5.3.6   Generate Ada Code

We encountered a number of problems (that we eventually resolved) while generating Ada code and parsing the generated code with REFINE/Ada (RA). Once we successfully generated and parsed Ada source code for the FCR subsystem, we generated Ada for the other subsystems, except for the SMS subsystem.

The Transformer component of the RET prototype copies the FORTRAN ASG and then transforms it into an Ada ASG. The first transformation rules that we wrote incrementally converted the FORTRAN ASG to an Ada ASG by performing a preorder traversal. This proved unsatisfactory because the earliest rules to fire destroyed information from the FORTRAN domain that was needed by subsequent rules. We switched to a postorder traversal to circumvent the problem.

Unfortunately, the n-ary rules for transforming multiplication, addition, subtraction, and division would not work with the postorder transform technique. This is because the rules, when applied to a node in the FORTRAN domain, would replace it with an Ada node that had FORTRAN child nodes. This was acceptable under a preorder traversal because transformation rule application resumes with the replacement node and continues downward, i.e., with the children. The children are then likewise transformed from the FORTRAN domain to the Ada domain. Under a postorder traversal, however, the children remain in the FORTRAN domain and are never converted to Ada.

We rewrote the rules to work under postorder traversals. The revised rules replace each FORTRAN node with another FORTRAN node that has Ada children. The non-exhaustive postorder traversal resumes with the replacement node, i.e., the FORTRAN node, and resumes upward. The children, being Ada nodes, need no further transformation.

We were unhappy with RA's Ada code formatting. The Ada printer generated with DIALECT and provided by RA doesn't support all of the formatting options listed in the SRS (Ref. 3). We understand that the Walnut Creek Ada CD-ROM* contains four Ada pretty printers. We did not examine the contents of the disk, but it is worth investigating whether one of the pretty printers produces more pleasing results than RA. We recommend reformatting the generated Ada code with an Ada pretty printer as a separate pass. This can be done without integrating the pretty printer into the RET prototype.

We were disappointed that RA doesn't support recompiling individual compilation units into the Ada library, as is permitted by the Ada language. We had envisioned that the RET Prototype would allow the engineer to generate Ada code, make some changes to one or more of the views, and then reanalyze only the affected compilation units. The RET prototype can regenerate the specific individual compilation units that the engineer selects, but the RA restrictions require the RET prototype to reanalyze *all* of the Ada code.

We found that the process of analyzing the generated Ada code for most of the subsystems required about half an hour or less for any one subsystem. This is short enough that editing the generated code to make small-scale changes and then reanalyzing it is feasible, but long enough that the changes are only practical if made in batches.

We separated the process of generating and analyzing Ada code into several steps to mitigate the inconvenience of RA's restriction on selective analysis. The engineer may begin the process after running the Packager and distributing the data objects. The engineer may then generate and browse *select* compilation units. The new Ada source code is not linked to the other views at this point because the RET prototype relies on analysis information that RA provides, but the engineer can still browse the views.

---

* The CD ROM contains various Ada software. It is available through Walnut Creek CD ROM, Inc., Suite 260, 1547 Palos Verdes Mall, Walnut Creek, CA 94596.

The engineer may check the syntax of generated code at any time by choosing a menu item that causes RA to parse, but not analyze, the *selected* compilation unit(s). The rest of the Ada code for the target system need not have been generated in order to parse any compilation unit. We separated this step from the others because RA parses code much faster than it analyzes code, and the engineer can thus find syntax errors that he may have introduced through manual editing without waiting for RA to analyze the entire target system.

The engineer may direct the RET prototype to generate Ada code for any *select* compilation unit by choosing the appropriate menu item. The engineer may thus generate Ada code for one unit without regressing any changes that he may have made through manual editing to a different unit.

The engineer may alternatively direct the RET prototype to generate, parse, and subsequently analyze Ada source code for all compilation units rather than select units. The engineer may find this easier if he has made many changes.

Separating Ada code generation from analysis allows the engineer to execute an Ada pretty printer, or other tools that read the generated Ada code, independently of the RET prototype. The RA parser recognizes the arrangement of whitespace, tabs, and newlines (or carriage returns) and stores the layout in the form of surface syntax. The RA printer that the RET prototype uses to display the Ada SCL view respects the surface syntax. If the engineer reformats the generated Ada source code before RA analyzes it, the SCL view will display the reformatted code.

We did not originally intend to support Ada code analysis in the RET prototype. The previous version of the ASRET Reengineering Process Model (Figure 1 in this report) that appears as Figure 6-3 in the *Software Requirements Specification of the Avionics Software Reengineering Tool (RET) Prototype System* (Ref. 3) indicates that the step labeled "Analyze Ada Code (Reverse Engineering)" is "Beyond the Scope of the Current Ada Project."

Separating the code generation and analysis steps allows the RET to support Ada code analysis as indicated in Figure 1. For example, the RET prototype produces the DFD from information contained solely on the RHS, and the RET could be modified to create a DFD from Ada source code for which no corresponding FORTRAN version exists.

6.   USING THE RET PROTOTYPE

The RET helps the engineer develop an Ada system by reusing parts of the existing system. It supports, but does not enforce, the ASRET process model by implementing macro and micro restructuring as defined in Section 3. The engineer first applies macro restructuring features to construct a skeleton of the Ada system. The skeleton provides the modular structure and the distribution of variables among the modules. The engineer then explores and refines the Ada structure, and adds program statements using micro restructuring features of the RET.

This section describes some problems that the engineer faces when reengineering a legacy system, i.e., a system that has undergone many modifications through years of maintenance, and explains how the RET helps the engineer overcome those problems. Section 6.1 describes the use of the *Packager* component of the RET. Section 6.2 discusses the use of the *Transformer* component and explains some of the issues involved in translating certain language features.

6.1   DEVELOPING PROGRAM STRUCTURE USING THE PACKAGER

The engineer learns about the FORTRAN system by navigating through several views. The engineer may be overwhelmed with information when initially confronted with an entire legacy system. One way to reduce the amount of information that the engineer must comprehend is to examine only the large-scale constructs of the program. For example, the engineer might first be interested in understanding the relationships between the modules that comprise the system. Then, he might turn his focus of concern to specific subsets (clusters) of the modules.

To discover the modular structure of the FORTRAN system, the engineer directs the Packager component of the RET to *cluster* subprograms into groups that will eventually become Ada packages. The Packager iteratively applies clustering techniques described by Hutchens (Ref. 18) and Müller (Ref. 17) to analyze the FORTRAN system and group subprograms based upon calling relationships and patterns of data usage, measured in terms of data bindings.

During the analysis, the engineer gains a better understanding of each subprogram's purpose and why subprograms are grouped as they are. The Packager invites the engineer to explore the most recently clustered modules after each iteration. The RET organizes the subprograms and aids the engineer in exploring groups of related subprograms that are, in the sense of the clustering criteria, more closely related than others. Note that understanding is a critical element of any software reuse or reengineering process, as we have noted on several research efforts which were predecessors to the ASRET project (Refs. 32-35).

6.1.1 Definitions

The Packager uses data bindings to cluster some modules. A *data binding* is a tuple (p, x, q) where p and q are subprograms that reference data object x. Hutchens (Ref. 18) describes several kinds of data bindings. The RET counts only *actual* data bindings, i.e., those in which the data object is written by one subprogram and read by the other. The RET analyzes actual data bindings because they provide a reasonable model that is not too computationally intensive to implement in an interactive tool.

The RET computes the Interconnection Strength (IS) and the Common Client and Supplier (CS) sets defined in (Ref. 17) based upon the actual data bindings. The RET alters Müller's IS metric slightly by adding one to it if either of the two subprograms calls the other.

Clustering produces a *tree of modules*. The root module represents the Ada library, intermediate modules represent packages, and the leaves represent subprograms. The root is defined to be at level zero and its children are at level one.

The Packager displays one graph for the Ada library, and one for each potential package. The Packager graph is an abstraction that enables the engineer to see relationships among program modules, such as data objects shared between them. *Nodes in the graph* represent nested modules, i.e., packages or subprograms, and the edges depict data binding relationships between them.

There is exactly one graph associated with any given nonleaf module, M, in the tree; we refer to it as *the* graph of M. The nodes in that graph correspond to the direct children of M in the tree. The edges between the nodes in the graph depict the data binding relationships between the corresponding packages or subprograms. We use the term *package structure* to refer to both the tree and its graphs. For simplicity, we refer to nodes as packages or subprograms or, when the distinction is unnecessary, as modules.

44

The graph in Figure 5 shows one subprogram *(FCR_OUTPUT)*, five packages *(fcr_df, fcr_dr, main, modes,* and *dead code)*, nine edges, and nine edge labels. An edge between two modules indicates that they share data bindings. An edge between a package and another module, M, indicates that at least one subprogram in the package shares data bindings with M. The edge labels list the data objects that comprise the bindings or show the subprograms in the package that are involved in the data bindings.

The edge labels in the Packager view provide information about the variables shared between the modules. The edge label between packages *modes* and *fcr_dr* lists the variable names involved in the data bindings between them. This label lists one variable *(IRSQ)* that is read and written by both packages, one variable *(MDR32J)* that is read by *modes* and written by *fcr_dr*, and ten variables that are read by *fcr_dr* and written by *modes*.

The edge label between *FCR_OUTPUT* and *fcr_dr* shows that there is one data binding between *FCR_OUTPUT* and each of seven subprograms nested directly under *fcr_dr* (i.e., *FCR_DR033* through *FCR_DR009*), and one data binding between *FCR_OUTPUT* and each of three subprograms *(FCR_AD0, FCR_DR003,* and *FCR_DR008)* nested directly under *fcr_ad,* which is nested directly under *fcr_dr*.
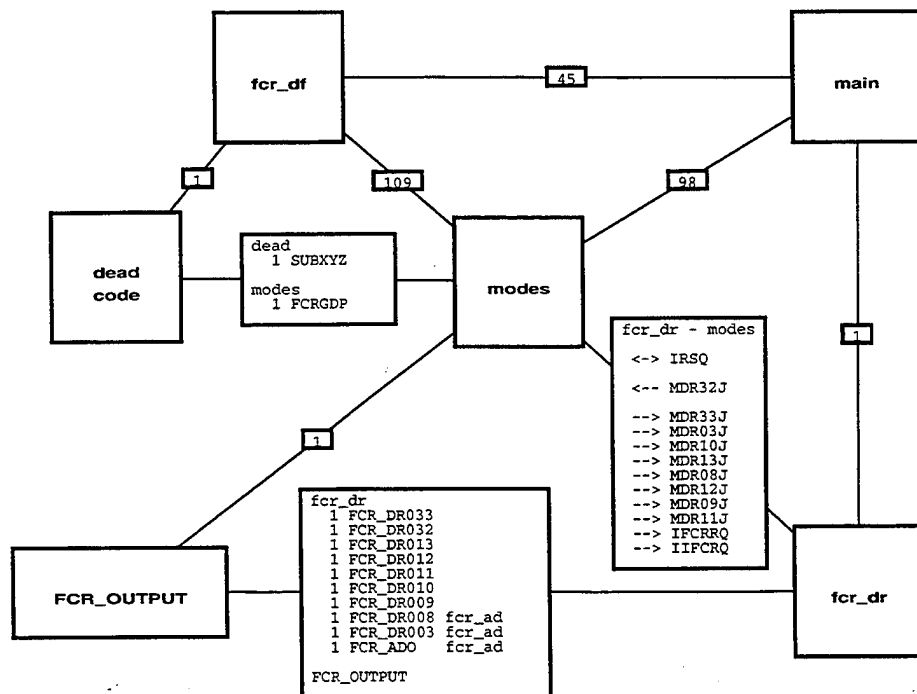

Figure 5   The Packager View

45

The label between *dead code* and *modes* is similar, except that it is between two packages. The label on the edge between *FCR_OUTPUT* and *modes* shows the total number of variables (1) shared between them.

6.1.2   Creating a Package Structure

Initially, every subprogram is at level one in the package structure and appears in the level-zero graph. The edges in the graph are not shown by default because there are generally too many of them, but the engineer may view (or hide) the edges adjacent to any module by choosing the appropriate menu option. The engineer may view the original source code associated with any module by selecting it with the mouse. The engineer may select specific subprograms to be grouped together, or apply an automatic clustering algorithm.

The automatic clustering algorithm is a variation of the *hierarchical agglomerative clustering* technique (Ref. 18). In the RET, the technique is extended to allow for manual alteration of the package structure. The RET provides two of the clustering metrics described in (Ref. 17). Each metric can be defined as a function of two subprograms.

The *Common Clients and Suppliers* (CS) metric counts the number of other subprograms that provide data to, or accept data from two subprograms. This metric is useful for locating and grouping utility or library routines, such as math or I/O routines. The *Interconnection Strength* (IS) metric counts the number of shared data items that two subprograms reference. This metric is useful in grouping subprograms that manipulate common global variables or exchange data by parameters.

The automatic clustering process is iterative. To begin clustering, the engineer normally selects an option to perform one clustering iteration using the CS metric. We call this strategy *CS-clustering*. It tends to group modules that receive data from, or pass data to the same modules. The Packager computes the common client and supplier sets for each pair of level-one modules and identifies the group of modules that share the greatest number of clients or suppliers. It then modifies the package structure to combine the modules in this group.

CS-clustering should be performed one iteration at a time for several reasons. CS-clustering only takes a few iterations to identify many of the utility subprograms, and the RET relies on the engineer to determine when to stop clustering. The RET also relies on the engineer to manually add or remove subprograms because the heuristic strategy is too simple.

46

Once the engineer judges that CS-clustering is not discovering any new utility subprograms, he may initiate *IS-clustering*. With this strategy, the Packager computes the interconnection strength between each pair of level-one modules; determines the maximum IS, denoted IS_{max}, among all the modules; and groups those modules that are involved in an IS_{max} relation. The engineer may perform IS-clustering one iteration at a time, but it is faster to direct the Packager to iterate until only one level-one module remains in the level-zero graph, i.e., all subprograms have been clustered into (possibly nested) packages.

The engineer should employ CS-clustering before IS-clustering because the former identifies groups of utility subprograms that are not recognized by the latter. If IS-clustering combined a utility subprogram with other subprograms, the CS metrics for the resulting package would be different from the utility subprogram CS metrics and the utility would be less likely to combine with other utilities during CS-clustering.

With either clustering strategy, when the Packager groups a set of modules, it creates a new level-one package and moves the grouped modules to level two, i.e., under the new package. Edges appear in the level-zero graph between the new package and any level-one modules that share data bindings with it.

With either strategy, the engineer may inspect and/or alter the package structure after each Packager iteration. Alternatively, the engineer may direct the Packager to iterate until every module has been included in some package, automatically providing an approximation to a reasonable package structure. The engineer must verify the resulting package structure or modify it through the views to obtain an appropriate grouping. The information provided by the Packager facilitates this analysis, and editing operations allow the engineer to easily change the structure.

The clustering strategies described above produce a hierarchical organization of packages; there are packages nested within other packages. Although the RET can generate Ada code corresponding to a hierarchical nesting structure, it may be easier to maintain Ada code which consists of smaller library unit packages because such designs tend to discourage redundancy and strengthen encapsulation. We expect that the engineer will often want to "flatten" the generated package structure, i.e., increase its width and decrease its depth. The RET generates *with* context clauses for any package that references a library unit.

The Packager tries to prevent the package structure from becoming unnecessarily deep by maintaining a threshold on the package size. When package A is to be moved into package B such that A would be nested within B, the Packager checks the number of modules in package A. If it is below the threshold specified by the engineer, then the modules in A are moved to B and the package A is eliminated.

This somewhat arbitrary heuristic is only useful for preventing the formation of many tiny packages and, in practice, the threshold must be set quite low. We have experimented with values of four or five. Nominally, we expect that the engineer will want to intervene during clustering and edit the package structure as it evolves in order to reduce nesting.

6.1.3 Editing the Package Structure

Packager graphs are interactive displays. The engineer opens pop-up menus by positioning the mouse cursor over a module, an edge, or the background or window title and clicking the right mouse button. The resulting pop-up menu shows commands for the module, edge, background, or window. We refer to this below as issuing a module, edge, background, or window command. The RET provides commands for *navigating*, *browsing*, and *editing* the package structure.

Navigation commands allow the engineer to display different graphs by clicking on a package or the background. The descend package command causes the RET to display a package graph. The ascend background command causes the RET to display the parent package graph.

Browsing commands alter the Packager display without changing the generated package structure. The RET prototype provides browsing commands on the module, edge, background, and window pop-up menus.

- Module commands allow the engineer to select or deselect individual modules or modules in a region, show or hide individual or selected modules, drag and reshape modules, and show FORTRAN and/or Ada source code.

- Edge commands allow the engineer to select or deselect individual edges, show or hide individual or selected edges, and show global or local bindings (or both) on edges.

- Background commands allow the engineer to arrange modules in a circle or grid, and refresh, scroll, and zoom the display.

- Window commands allow the engineer to move, refresh, hide, reshape, and close the Packager display.

Editing commands allow the engineer to edit module names and alter the Ada package structure. The engineer may move a module from the current graph to another package in that graph via the push command, or to the parent package graph via the pop command. The pop-to-top command moves a package all the way up to the Ada library level. The disperse command eliminates a package from the current graph and moves all of the modules that were nested in it up one level in the graph. The RET maintains the edges between the packages and subprograms as the engineer changes the package structure.

The engineer can assign navigation and source code display commands to the middle mouse button to reduce the number of mouse or keyboard events required to effect a command. We have found this to be very convenient when working with a large system. The left mouse button is always assigned to the select and deselect commands. The right mouse button is always assigned to the pop-up-menu command.

6.1.4 Distributing Data Items

The Packager automatically distributes global data items among the modules of the package structure. The algorithm reduces each data item scope while maintaining its visibility as needed. It is based upon the following criteria.

- If a data item is used only by subprograms in a single package, the data item declaration is placed in the package body.

- If a data item is used by subprograms in more than one package, but most often by subprograms in a particular package, the data item declaration is placed in that package specification. Other packages that use the data item specify a context clause for the package.

- A new package is created for each common block with remaining undistributed data items. These data items are used by subprograms in more than one package, with no package clearly using them more often. The data item declarations are placed in the new package specifications, and other packages that use the data items specify context clauses for the new packages.

The data object distribution algorithm is most effective when there are many data objects declared in one module, such as a common block, that are referenced by few other modules. Embedded systems may use common blocks to map variables to specific memory locations. The FCR system, for example, has one common block with 396 variables that are memory mapped. Distributing these variables among the packages so as to reduce the scope of their declarations would disperse the specification of the mapping throughout the code and make it more difficult to change the memory mapping (say, in response to an upgrade to the processor that doesn't preserve the original address).

6.2 TRANSLATING PROGRAM STATEMENTS USING THE TRANSFORMER

Once the engineer has constructed and refined the package structure of the Ada system and placed the variable declarations where desired, the Transformer helps with micro restructuring by translating individual statements from the source to the target programming language. The engineer may inspect the FORTRAN Source Code view for any module and select specific statements with the mouse. The Transformer translates these selections to Ada and inserts them into the Ada ASG. If the engineer has renamed variable declarations, then the RET prototype generates references to those variables.

The Transformer generates Ada code for the package structure that the Packager produces. First the Transformer creates a skeleton of the Ada system, and then it transforms individual statements. The skeleton Ada code comprises package and subprogram specifications and bodies that may include variable and constant declarations. The subprogram bodies include a single *null* statement. The Transformer generates subunits as the engineer specifies. The transformer also generates a type package that defines all of the types and subtypes referenced in the variable declarations. The type package declares Ada types that correspond closely with FORTRAN types, although an exact mapping is not generally available as explained below.

Translating FORTRAN statements that map readily onto Ada language features is straightforward. For example, the Transformer can easily translate *Block IF* and *DO / END DO* statements into Ada IF and LOOP statements, respectively, because the semantics are consistent between the languages. The fact that the control variable of a FORTRAN *DO* statement remains defined after the loop is a nuisance. For these structures, we needed to account for differences in the way they are represented in the ASGs, but did not need to implement further analyses.

There are FORTRAN constructs for which the mapping to Ada is not obvious or for which there is a choice of translations. We have found several sources of difficulty in transforming individual statements in such a way as to avoid propagating undesirable FORTRAN constructs while taking advantage of Ada language features not present in FORTRAN. They include the use of *unstructured control constructs*, the general lack of correspondence between *language features* and, in particular, differences in the *data type systems*.

50

Unstructured Control Constructs — Some FORTRAN code contains unstructured control forms, defined simply as branches into or out of loops or decisions (Refs. 36, 37). While such forms do not *always* impede maintenance, they usually make the code harder to understand and modify. Unstructured control forms exist in code which was written before the benefits of structured programming (Ref. 38) were widely acknowledged.

Some FORTRAN language features encourage unstructured designs. *Arithmetic IF* statements cause control to be transferred to any one of three locations based on a test. *Logical IF* statements are only problematic when they are used with *GOTO* statements. VAX FORTRAN extended ranges (in *DO* loops) are egregious examples of unstructured constructs which might effectively confound maintenance programmers. We would not want to reproduce such a design in the Ada system and, in any case, restrictions on Ada *GOTO* statements (Ref. 39) preclude using them to transfer control into an Ada *LOOP*.

We implemented the Preliminary Restructurer (PR) component of the RET prototype by running a commercial control flow restructuring tool, SPAG (Ref. 26), to eliminate control structures that are difficult to translate. The tool removes most of the objectionable constructs. We apply the tool in a preprocessing step so that the RET may assume that certain structures are not present in the FORTRAN source code.

Without such a tool, the presence of constructs such as those discussed above would have forced us to implement our own control flow analysis to avoid generating Ada code that is as difficult to understand as that in the FORTRAN system. Although we identified formal techniques for control flow restructuring (Refs. 36, 37, and 40) in our literature survey, we found it less costly to apply the SPAG component of plusFORT.

Language Features — The RET prototype generates code for Ada language features that have no counterpart in FORTRAN, but which produce programs that are substantially easier to maintain. For example, the RET does take advantage of Ada packages because we feel that they are useful for encapsulating code and help to reduce the ripple effect (Ref. 41) of modifications. On the other hand, the RET prototype does not generate Ada code which uses exceptions because we believe that they make the code more difficult to understand and, except in select situations, are of limited value in simulation software.

Data Type Systems — FORTRAN has fewer types than Ada and it allows implicit conversions which must be made explicit in Ada. Data types that seem to serve the same purpose may have different implementations across languages. The application may even rely upon compiler implementation details or undocumented language features. This fact is often an important consideration when translating embedded systems. The example in Section 7.1 illustrates some complications we faced in converting data types.

6.3   A SAMPLE APPLICATION

During the Reengineering Tool Development task, we exercised the Packager on the FCR subsystem of the F-16 OFP simulation code. The FCR subsystem comprises 98 subprograms and 17,641 lines of code as measured by the Unix *wc* utility. Section 5 describes the analysis that we performed on the F-16 OFP simulation code to exercise the RET prototype during the Reengineering Tool Testing and Evaluation task.

We found that by first performing a few CS-clustering iterations, we were able to identify groups of subprograms that simply assigned values to related sets of data. In subsequent IS-clustering iterations, we grouped subprograms that perform processing for the various FCR modes. We grouped the remaining subprograms based upon their calling structure and identified the overall organization of the system. We examined the calling structure by viewing the Call Diagram, automatically generated by the RET for the FORTRAN system.

The FCR code has three common blocks containing 422 global variables. We directed the Packager to automatically distribute those global variables throughout the packages. The Packager moved 333 variable declarations into packages and created three new packages, one for each common block, to hold the remaining global variables.

We recognized by browsing the include files that the variables in one of the common blocks were memory-mapped via *EQUIVALENCE* statements. Under the assumption that every variable for which such an equivalence exists is memory-mapped, we moved them to their own data package. The validity of our assumption could easily be verified, or alternative clustering heuristics could be suggested, by maintainers familiar with the FCR code. These individuals were not available during this phase of the RET prototype development.

7.   RET PROTOTYPE IMPLEMENTATION

This section describes both characteristics of the RET prototype implementation that involve tradeoffs and flat limitations. The former result from decisions that evolved during prototype development in recognition of the scope and goals of the ASRET project. The latter simply reflect features that we did not implement due to resource constraints and the current state of the practice in language reengineering technologies. The absence of any capability in the current RET prototype should not be construed as discommendation. We believe that the deferred capabilities are worth pursuing in a next generate of the tool.

7.1   IMPLEMENTATION CHARACTERISTICS

Type Deduction — VAX FORTRAN provides a *LOGICAL* data type to represent the boolean values *.TRUE.* and *.FALSE.*. It also provides *logical operators* that operate on values of that type. Some examples of the logical operators are *.AND.*, *.OR.*, and *.XOR.*. At the machine architecture level, the least significant bit determines a LOGICAL value and the other bits are undefined. The VAX FORTRAN language (Ref. 31) guarantees that the logical operators affect the least significant bit in variables that are declared type LOGICAL. A particular compiler, however, may implement the logical operators such that they affect all bits.

Depending upon the compilers, it *may* be possible to define an Ada type which reproduces the behavior of the FORTRAN LOGICAL type. For example, an Ada type may be derived from the predefined enumeration type *BOOLEAN*. If an enumeration representation clause is provided and the appropriate size specification is given in a length clause, then the Ada logical operators *and*, *or* and *xor* may be substituted for the corresponding FORTRAN logical operators.

An embedded system which uses memory-mapped I/O may take advantage of compiler implementation details that are not specified by the language. If the FORTRAN logical operators affect all bits of LOGICAL variables, and if the application relies upon a compiler implementation that operates on all bits, then converting LOGICAL to BOOLEAN will not suffice to reproduce the behavior of the original program because the bit patterns in memory will be different.

The VAX FORTRAN logical operators also may operate on variables of type INTEGER, in which case they operate on all corresponding pairs of bits. This is convenient for implementing memory-mapped I/O in FORTRAN. The RET must not translate FORTRAN variables of type INTEGER into Ada variables of any integer type if they are to be used in logical expressions because the Ada logical operators are only defined over type *BOOLEAN*.

The VAX Ada package *SYSTEM* defines subtypes of positive integers that are intended to be used in bit operations. It also overloads the Ada logical operators for these bit subtypes so that they have the same effect on memory as the FORTRAN logical operators (when applied to INTEGER variables). The bit types defined in the VAX SYSTEM package would seem to be a suitable replacement for the FORTRAN INTEGER type, except that FORTRAN INTEGER variables may be used as signed integers in arithmetic expressions as well as in logical expressions. Although the VAX Ada bit subtypes are integers, they are unsigned.

The RET could have defined a new type, derived from universal integer, that includes negative integers and overloads the logical operators. Transforming *all* INTEGER object declarations in the FORTRAN system to Ada object declarations that reference the new type would simply propagate the loose typing to the target system. This would be an unfortunate result when the target system has a very complete type system as does Ada.

Instead, the RET takes advantage of the fact that while a FORTRAN integer variable *may* appear in both arithmetic and logical expressions, it *isn't likely* to. The RET only declares such a variable as an Ada bit subtype if it is used in logical expressions, but not in arithmetic expressions.

Actually, there are two forms of arithmetic expressions which are allowed for bit subtypes. They are $V*C$ and V/C, where V is a variable and C is a constant that is a power of two. In our sample FCR FORTRAN code, no variables are used in both logical expressions and in disallowed forms of arithmetic expressions. The resulting code is easier to understand because the type marks correctly indicate whether the variables contain bit fields or integers.

Transforming Comments — We were not very pleased with the way that REFINE supports comments. REFINE/FORTRAN attaches comments to the (LHS) ASG as text fields that Reasoning Systems refers to (Ref. 21) as "surface syntax." REFINE/FORTRAN

frequently attaches the surface syntax to the parent of the (LHS) ASG node that it describes. The Transformer recognizes this and attempts to copy the surface syntax to the appropriate node in the RHS ASG. Unfortunately, REFINE/Ada moves the surface syntax back up to the parent of the node in the RHS ASG when it analyzes the generated code.

In spite of this problem, the RET prototype does transform FORTRAN comments into Ada comments. We generated Ada code for the FCR subsystem that apparently contains all FORTRAN comments present in the FORTRAN FCR code. The RET always places the comments in the right location in the generated Ada code.

Hidden Implementations — The RET prototype implements some of the views defined in the Software Requirements Specification (Ref. 3) indirectly or in nonobvious ways. The RET prototype does not directly provide a LHS DFD, RHS DED, or RHS CD. The information in the LHS DFD is present in the Packager view. The information in the RHS DED is also present in the Packager view. Specifically, the Packager lists all packages and subprograms and a window that the engineer can pop-up from the Packager view shows the data objects defined in the selected package. The calling information in the RHS CD is presented in the call edges of the Packager view.

7.2 LIMITATIONS OF THE RET PROTOTYPE IMPLEMENTATION

We recognized three possible approaches to prototyping the RET during the Reengineering Tool Development task. The first was to systematically identify all FORTRAN language features and develop capabilities to address specific features. The second was to devise and implement individual capabilities, each targeted towards specific elements of a formal FORTRAN syntax. The third was to devise transformations as needed for a sequence of FORTRAN programs, chosen to progressively introduce more and more elements of the FORTRAN language.

The approaches based upon systematically identifying FORTRAN language features or enumerating elements of a formal syntax seem attractive because they promise a means to measure the coverage of the reengineering capabilities with respect to the FORTRAN language. A number of considerations belie the apparent expedience, however.

FORTRAN became popular before formal language theory was well established and the REFINE/FORTRAN grammar does not capture every aspect of the language. The formal grammar does not represent the semantics or pragmatics of the language. The sequence of states of an avionics simulation system, i.e., the bit configurations of memory-mapped variables and the state of the machine in general is an essential characteristic

that the target system must reproduce. The target system must do much more than just reimplement the logic and formulas encoded in the subject system. *The essential behavior is sometimes dependent on the compiler implementation.*

A list of language features, such as the VAX FORTRAN LRM (Ref. 31), is a canonical organization invented to preserve and arguably to transfer knowledge on the syntax and semantics of the language. We did not set out to organize the transformations according to the LRM because we didn't think the structure would help us conceive the transformation techniques or implement the rules. Indeed, we found that the transformations are highly order dependent, and the order is prescribed by the techniques and follows from the implementation details and practical considerations. We did attempt to cover the language features enumerated in the FORTRAN LRM when developing the transformations, but we didn't follow the LRM as a guide to developing the RET.

We viewed the third alternative as the most practical given the great uncertainty that we faced in applying the selected methods to reengineering. We started by writing transforms for code fragments and then small subprograms. We later built more transforms and translated the subprograms of the FCR subsystem. We then modified the transforms to overcome type conversion problems and allow REFINE/Ada to parse and accept the target system. Eventually, we translated the other subsystems and once again corrected the Transformer as needed to parse the target code.

A benefit of this spiral-model approach is that we did not develop a large body of transforms only to find out that the predetermined strategies would not work. On the contrary, each transform was written to transform a particular sample of the subject system. One limitation of this approach in prototyping is that the RET now supports only those language features that are employed by the FCR, UFC, MFD, GPS, INS, and RLT subsystems. The RET prototype does not support the FORTRAN language features listed in Table 6, and described in the indicated sections of the VAX FORTRAN LRM (Ref. 31).

Limitations in Ada code formatting due to the DIALECT printers are discussed in Section 4.2. The RET prototype does not provide the following capabilities, which are defined in the RET SRS (Ref. 3).

- DFD Anomaly Detection
- DFD Data Store Nodes
- Hypertext Annotations.

56

Table 6 Unsupported FORTRAN Language Features

FEATURE DESCRIPTION	VAX FORTRAN LANGUAGE REFERENCE MANUAL SECTION
COMPLEX types and constants	2.2.1
INTRINSIC statements	4.9
NAMELIST statement	4.10
RECORD Statement	4.13
SAVE Statement	4.14
Structure Declaration Block	4.15
VOLATILE Statement	4.16
PAUSE Statement	5.8
STOP Statement	5.10
ENTRY Statement	6.2.4
I/O Statements	7, 8, 9
Compiler Directives	10

The level of effort for writing transformations that produce correct Ada syntax was roughly consistent with our predictions. The effort involved in enhancing the transformations so that the REFINE/Ada parser and analyzer would accept the generated code was greater than we expected. We did not implement all of the features of the RET prototype that we had originally planned, primarily because we spent more time on the Transformer component than we expected.

Most of the Transformer problems arose from the great differences between the FORTRAN and Ada type systems, the type perversions that FORTRAN supports in the name of programming convenience, and our goal of reproducing sequences of bit patterns for memory-mapped variables where necessary, as described in Section 7.1.

8.   RET PROTOTYPE PLATFORM

The RET prototype runs on a Sun SPARCstation 10/40 under the Sun OS 4.1.3 Unix operating system. The ASRET hardware includes a SPARCclassic X-Terminal in addition to the SPARCstation console. The engineer may run the RET prototype from either seat. Table 7 provides the release numbers for the software products installed on the SPARCstation hard disk. The remainder of this section documents the RET prototype software platform including the directory structure and files on the hard disk.

Table 7 Installed Software Product Versions

VERSION	PRODUCT NAME
3.1	REFINE
1.1	INTERVISTA
1.0	DIALECT
1.2	REFINE/FORTRAN
1.0	REFINE/Ada
4.5.2	plusFORT (SPAG)
X11R5	X-Windows
18	GNU Emacs

8.1   SOFTWARE FILES

In all references to files that begin with a tilde (~), the tilde represents the directory /usr/dew.

Reasoning Systems Files — The Reasoning Systems software products are installed in /usr/local/reasoning. This directory contains the loadable code and data files for Software Refinery, REFINE/FORTRAN, and REFINE/Ada. These products may be loaded after starting REFINE, but it is inconvenient to load the individual products on a daily basis. We created a "REFINE image" that the engineer can alternatively load to quickly establish a REFINE session after logging in.

The process of creating the image consists of starting REFINE, loading the products, compacting or forcing garbage collection, and saving the session to a binary file, as explained in the REFINE User's Guide (Ref. 21). We needed to create the image from the SPARCstation console, not the SPARCclassic X-Terminal, to make it usable from either seat. We also needed to set the *refine-start-switches* variable by entering the command

```
(setq   *refine-start-switches*   (append   *refine-start-switches*
'("-qq"))) into the *scratch* Emacs buffer before starting REFINE as instructed in the
```
release notes (Ref. 42).

We begin every login by executing the `initialization-script` file (Table 8-2). The REFINE image may be loaded by entering the "`M-x run-a-refine`" command in Emacs and replying "/usr/local/reasoning/asret/all.sys," the name of the REFINE image file that we saved. It is not necessary to recreate the image file to run the RET prototype.

After loading the image, we load the RET by cutting the following commands from the file `~/asret/refine-commands.re` and pasting them into the *REFINE* buffer:

```
(progn
  (setq *default-pathname-defaults* "~/asret/ret/")
  (setq ada::*null-library-source*
    "/usr/local/reasoning/asret/dev/data/standard.ada")
  (load "~/asret/ret/initialize-session")
  (in-package 'scp))
```

The `standard.ada` file defines the Ada package standard for REFINE/Ada. We added the following VAX Ada extensions from Appendix F of the VAX Ada LRM (Ref. 43):

```
type UNSIGNED_BYTE is range 0 .. 255;
for UNSIGNED_BYTE'size use 8;

function "not" (LEFT      : UNSIGNED_BYTE) return UNSIGNED_BYTE;
function "and" (LEFT, RIGHT : UNSIGNED_BYTE) return UNSIGNED_BYTE;
function "xor" (LEFT, RIGHT : UNSIGNED_BYTE) return UNSIGNED_BYTE;
function "or" (LEFT, RIGHT : UNSIGNED_BYTE) return UNSIGNED_BYTE;

type UNSIGNED_WORD is range 0 .. 65535;
for UNSIGNED_WORD'size use 16;

function "not" (LEFT      : UNSIGNED_WORD) return UNSIGNED_WORD;
function "and" (LEFT, RIGHT : UNSIGNED_WORD) return UNSIGNED_WORD;
function "xor" (LEFT, RIGHT : UNSIGNED_WORD) return UNSIGNED_WORD;
function "or" (LEFT, RIGHT : UNSIGNED_WORD) return UNSIGNED_WORD;

type UNSIGNED_LONGWORD is range MIN_INT .. MAX_INT;
for UNSIGNED_LONGWORD'size use 32;

function "not" (LEFT   : UNSIGNED_LONGWORD) return UNSIGNED_LONGWORD;
function "and" (LEFT, RIGHT : UNSIGNED_LONGWORD) return UNSIGNED_LONGWORD;
function "xor" (LEFT, RIGHT : UNSIGNED_LONGWORD) return UNSIGNED_LONGWORD;
function "or" (LEFT, RIGHT : UNSIGNED_LONGWORD) return UNSIGNED_LONGWORD;
```

We set several options in the REFINE/FORTRAN initialization file (~/.refine-for-tran) by specifying the following statements in it. The RET prototype assumes that the REFINE/FORTRAN analysis was performed with these options set.

```
(setf *RETAINING-AST* 'T)(setf *SAVE-ANALYSIS* 'T)
(setf *DISPLAY-PERFORMANCE-ONLINE?* 'NIL)
(setf *DO-SET-USE-ANALYSIS* 'T)
(setf *COMPUTE-TRANSITIVE-WHO-CALLS-WHO* 'T)
(setf *KEEP-CONSTANTS-EVEN-IF-NOT-SET-OR-USED* 'T)
(setf *DO-RECORD-PARAMETER-DEFINITION-AS-SET* 'NIL)
(setf *DO-COMPLEX-ARGUMENT-USE-PROPAGATION* 'T)
(setf *USE-DEEP-SET-USE-FOR-CALL-ARGUMENTS* 'T)
```

Revision Control System Files — The Revision Control System (RCS) is installed in ~/gnu/rcs5.6.0.1. All of the RET prototype REFINE source code files are maintained in the RCS directory /usr/local/reasoning/asret/dev/src.

GZIP Files — GZIP is a freeware compression program installed in directory ~/gnu/gzip-1.2.4. We used it to install RCS.

SPAG Files — The plusFORT product from Polyhedron Software Ltd. is installed in directory /usr/local/reasoning/asret/plusFORT. We made several modifications to the SPAG configuration file to tailor processing as required for the SCP component of the RET prototype.

Development Working Directory — We developed the RET prototype by checking the REFINE source code files (called .re files) out from RCS and placing them into the ~/asret/ret working directory. Loading the initialize-session file noted above causes REFINE to load the source code files and compile them resulting in files with the same names, but ending in the suffix .fasl4. We then start the RET prototype by evaluating the form (ret).

The development directory /usr/local/reasoning/asret/dev/ contains several subdirectories:

- **data:** REFINE/Ada definitions of the Ada package standard
- **log:** backup log files created by the backup commands listed below
- **ref:** reference versions of the RCS files
- **src:** RCS source code directory containing the .re files
- **bin:** various scripts and commands listed below

60

The commands in `/usr/local/reasoning/asret/dev/bin` are listed in Table 8.

Table 8 Commands and Scripts Used During RET Prototype Development

COMMAND/SCRIPT	DESCRIPTION
aci, faci	check in a file and update the reference copy and the associated .fasl4 file
aco, faco	check out a file and copy the associated .fasl4 file
acou	check out a file without locking it
acou-all, facou-all	get copies of all files without locks, including the associated .fasl4 files
arcs	issue the rcs command
backup*, bkup*, and restore	various backup and restore scripts
include-files-to-lower-case, include-files-to-upper-case	used to rename include files
initialization-script	define environment variables and aliases
insert-headers.el	elisp (Emacs Lisp) file used to edit .re files
rcs-header.re	template for the RCS data in .re files
unlock	script to unlock RCS files

8.2   DATA FILES

We created the data files listed in Table 9 while exercising the RET prototype during the Reengineering Tool Testing and Evaluation task.

Table 9   RET Prototype Evaluation Data Files

DIRECTORY	DESCRIPTION
~/asret/ada	small sample ada programs
~/asret/fortran	small sample FORTRAN programs
~/asret/temp/ada	RET prototype generates Ada code to this directory
~/block40/original-source	original FORTRAN source code
~/block40/spag-source	FORTRAN source code processed by SPAG
~/block40/spag-source/ada	archive of Ada code that we generated with the RET prototype
~/fcr/original-source	the FCR subsystem
~/fcr/spag-source	the FCR subsystem processed with SPAG

61

9. CONCLUSIONS AND RECOMMENDATIONS

9.1 PROJECT STRUCTURE SUMMARY

For the Software Reengineering Study task, we conducted an extensive investigation of software reengineering tools and methods. We studied software reengineering tools and methods relevant to ASRET and recorded the information for use in the subsequent tasks.

During the Reengineering Process Model Development task, we developed a reengineering process model based upon the results of the first task, and developed the software requirements for the RET prototype.

In the Reengineering Tool Development task, we designed and implemented the RET prototype and exercised it by transforming part of the Fire Control Radar (FCR) subsystem from Block 40 of the F-16 OFP simulation software.

We executed the RET prototype and converted all but one of the subsystems in Block 40 FORTRAN code to Ada in the Reengineering Tool Testing and Evaluation task.

9.2 CONCLUSIONS

For the Literature Survey, we conducted a broad review of the state of software reengineering. We identified existing reengineering tools, software products, and techniques that that we thought were both relevant to ASRET and implementable. We screened and organized reengineering literature to distinguish the most promising methods consistent with the ASRET objectives. We were most interested in discovering results that had been demonstrated and proven effective in improving software.

An important lesson that we learned during the study is that the motivations, activities, and results that characterize the state of the practice in reengineering are not homogeneous. The field is bound by no more specific common interests within the community than is, say, engineering. The implication is that any successful reengineering effort must be predicated on a particular class of problems and the nature of the work must be tailored to the specific problem domain. A reengineering solution must be directed at the root cause of the problem to be effective, and must not just seek to alleviate the symptoms.

We discovered no broad spectrum reengineering nostrum for improving software. Every method that we investigated involved tradeoffs. We resolved, then, to focus the RET prototype development effort and restrict its scope based upon the *primary* needs of our sponsor. The RET prototype would be a language translation aid that automates as much of the job as is practical, leaving the rest to the engineer.

With that goal in mind, we developed a reengineering process model that specifies a sequence of tasks to reengineer a program written in FORTRAN to Ada. The reengineering process model includes modern software development processes, such as continuous testing, iterative restructuring and redesign, and configuration management. The engineer performs redocumentation and macro and micro restructuring and redesign steps multiple times, building upon the results of the previous pass during each iteration.

We developed a RET prototype that helps the engineer develop an Ada system by reusing parts of the existing system. It supports, but does not enforce, the process model by implementing macro and micro restructuring. The engineer first applies macro restructuring features to construct a skeleton of the Ada system. The skeleton provides the modular structure and the distribution of variables among the modules. The engineer then explores and refines the Ada structure, and adds program statements using micro restructuring features of the RET.

We built up the RET capabilities incrementally to mitigate risk by devising transformations as needed for a sequence of FORTRAN programs, chosen to progressively introduce more and more elements of the FORTRAN language. We found that the transformation rules are highly order dependent, and the order is prescribed by the transformation techniques and follows from the implementation details and practical considerations. The incremental approach allowed us to contain the impact of such discoveries to the most recently developed components.

We are convinced after exercising the RET prototype that the ASRET process model is sound and the top-down reengineering style that it encourages is effective. One of our goals for the RET prototype was that it help the engineer capture as much of the existing design from the original program as possible, and the RET accomplishes this.

A successful reengineering effort must be predicated on a particular class of problems and the nature of the work must be *tailored to a specific problem domain*. A reengineering solution must be *directed at the root cause of the problem* to be effective, and must not just seek to alleviate the symptoms. These two very important points merit further discussion because they are often ignored in reengineering efforts.

The root cause of the problem may be, for example, that the legacy system is so difficult to maintain that enhancements to provide additional capability are impossible because corrective maintenance consumes all the resources. This is a very different situation than, say, a rehosting effort impelled by the fact that the hardware platform is obsolete and parts are no longer available, or the software development environment is no longer supported; here time is of the essence.

The first problem could be addressed by revisiting the design, and perhaps even the requirements for the system, and generally requires substantial time and labor. The second problem may take advantage of a more automated approach, leaving the overall design intact and focusing on the implementation. Porting the first legacy system to a new environment while preserving its decrepit design achieves no benefits at great cost. Taking time to redesign the second application could result in a crisis before it is implemented.

The work must be tailored to a specific domain. The principles embodied in the RET prototype and the techniques that it implements are extensible to other languages, but we are now aware that it is no simple matter to change the RET to translate between other source and target languages. Many of the technical challenges are borne of fundamental differences between the source and target programming language models and implementation details, such as those between the FORTRAN and Ada type systems.

The programming language dependencies are not isolated to RET front and back ends, but are systemic and essential to the transformation strategies that it employs. This serves as a metaphor for the multifarious nature of reengineering in general; it represents the underlying reasons why it is so important to select reengineering techniques appropriate to the project at hand. It also controverts development of general purpose reengineering tools for broad markets and is consistent with the lack of a standard tool or abstraction, or a universal formula for success. The state of the practice is to craft custom tool suites and develop proficiency in them within the organization.

The Task 1 study revealed that developing the techniques for dissecting legacy systems is difficult, so in Task 3 we were not surprised by the moderate level of comprehension that we *initially* achieved through the RET views, the resistance of some subsystems to clustering via the Packager, the complexity of some Dataflow Diagrams, or the number of type conversions in the generated Ada code. While immersed in the development process, we accepted these with the understanding that even moderate advances are useful.

As we explored the F-16 simulation system during Task 4, relieved of the pressure to produce code and free to explore what we had already created, we began to see past the design tradeoffs and implementation constraints and we gained a greater appreciation of the potential power of the interactive views and transformation capabilities of the RET prototype.

Once we stepped back from the RET development effort and the mechanical *process of composing* the RET views and studied the output, we more fully realized the utility of the *information they provide* and recognized that they are of great practical value despite the technical imperfections that we grappled with in Task 3. We found that they summarized salient features of the subject system and focused our attention on key areas, while providing information that is not directly accessible from the source code.

One of the goals of the ASRET project was to research and develop a RET *prototype* for avionics support software reengineering. *The RET prototype that we have developed has the potential for reducing the resources needed to reengineer avionics support software. It would help human engineers produce more maintainable systems if it were developed into a product.*

We built the prototype to examine promising reengineering techniques. We identified relevant techniques in Task 1 and we selected the ones that we thought were implementable and efficacious in Task 2. The techniques that we selected had been demonstrated in various prototypes, but there was little or no hard evidence available to *prove* that they were effective in improving avionics support software.

Our experiences in exercising the RET prototype to translate the F-16 OFP simulation code during Task 4 convinced us that the RET can be effective. It remains for independent practitioners to test this conjecture and to help focus continued technology insertion efforts in areas that are likely to yield the greatest payoff.

We discuss important aspects of the RET prototype implementation and our experience with the RET throughout this report. We suggest numerous alternatives for improving the RET while leaving open other questions because more research is needed. We review the most significant of these issues below.

While the Packager generally works well, some of the subsystems that we analyzed were resistant to the RET's clustering techniques and demanded more of our attention. We encountered some large sets of subprograms that were so tightly coupled that we discerned no clear component structure. Additional research might seek to determine whether splitting subprograms in these situations would lead to a larger number of cohesive clusters with less coupling, or to identify how sensitive the Packager's effectiveness is to the engineer's experience and knowledge of the application.

The RET prototype generates Ada source code that the REFINE/Ada parser and analyzer accept. The Transformer could be more robust. The generated Ada code must be run through an Ada pretty printer. Deficiencies in Ada code generation are easily circumvented by editing the source code before REFINE/Ada parses it. With very little help from the engineer, the RET prototype generates almost all of the Ada source code needed to re-implement the subject system.

We are concerned about the large number of type conversions in the target system, but we suspect that the number can not be greatly reduced because of the inherent differences between the FORTRAN and Ada type systems. We didn't spend much time overriding the data type deductions made by the RET prototype during the Reengineering Tool Testing and Evaluation task, but it is possible to do so. We expect that an engineer would question some of the type decisions that the RET prototype made. It would be interesting to examine how many of the type conversions could be eliminated by overriding types or making modifications to the source code by hand.

Some of the DFDs that the RET prototype generated are *extremely* complicated. We would like to know if those complications reflect inherent limitations in the DFD or an unfortunate subject system design. If they signal application areas that require more human intervention during reengineering, then even the incomprehensible graphs in the DFD serve a useful purpose and the DFD concept doesn't necessarily need to be reconsidered. If the corresponding parts of the applications are well designed yet necessarily complex, then we would judge that the DFD poorly represents those parts.

We do not argue that the overpopulated DFD graphs are actually desirable. The ideal representation is one that decomposes even the most horrendous application into simple and clear abstractions in which the base processes are manifest. The DFD doesn't achieve the ideal; only some of the graphs are clear. The unresolved question is whether the unclear graphs convey a useful message about the remainder of the subject system,

viz. that the design is unacceptable. Specifically, should we develop DFD complexity metrics to predict risk or guide decisions such as where to focus additional resources during reengineering?

We would like to exercise the RET prototype on other applications, perhaps in different application domains to investigate whether the DFD views are more appropriate to certain classes of applications. The simulation code for Block 40 of the F-16 OFP contains a very large common block that is shared by almost every subprogram in the system, while parameters are rarely passed. Not all applications share these design characteristics. We would like to compare the DFDs that we created for the Block 40 code with DFDs representing applications where data is primarily exchanged via parameter passing rather than through shared memory.

We had very little trouble with Software Refinery environment or tool anomalies, e.g., abnormal termination. The RET provides a stable reengineering environment. We were not surprised by the lack of modern GUI features available through INTERVISTA because we investigated the product before selecting it, but we were nevertheless disappointed that we could not create a more sophisticated user interface for some features. This is not a problem for the prototype, yet users may perceive limitations if the RET is developed into a product. The RET should be updated to take advantage of enhancements provided in current releases of INTERVISTA.

Limitations inherited from INTERVISTA include the lack of text clipping in the hardcopy versions of the graphical views, and the lack of control in specifying the absolute size or scale of the hardcopy views. There are too few icon styles available to distinguish different kinds of nodes. For example, the package and subprogram nodes in the Packager view are indistinguishable except by the labels that the RET prototype adds. Nevertheless, the content and substance of the interactive graphical views outweigh any defects in the static representations.

The DFD and Packager are complementary views that display *some* similar information. While exercising the RET prototype, we found it useful to examine both views. The RET prototype creates the DFD from the generated Ada code which means that the engineer must work with the Packager view to generate Ada before the DFD view is available. We think that an engineer might find an LHS DFD view useful while building the Packager view.

9.3 RECOMMENDATIONS

How do we prove that the RET prototype is valuable? Hypothetically, we could perform an experiment to evaluate the tool where the test project employs it and a simultaneous "control" project doesn't. The maintenance costs for the resulting parallel target systems over several years might correlate with the RET's effectiveness. After all, isn't measuring the desired variable (cost) directly more reliable than observing possible indicators?

A fundamental defect in this approach is that uncontrollable, if not unknown, variables are at play. We could not, for example, objectively adjust for differences in the staff's experience, technical and managerial talent, methodology, or culture. Correlation of the use of the RET with cost would not imply causality. A great practical impediment is that such luxurious "fly-offs" are rare because reengineering plans are motivated, at least in part, by an anticipated reduction in the cost of the program rather than an increase.

We recommend instead that software maintainers participate in an experiment, i.e., a Beta test, using an enhanced RET to reengineer an application in a production environment rather than in a laboratory. The RET is a prototype that we developed under AS-RET to evaluate reengineering technology and it needs some improvements before production software maintainers who are not experts in language processing could achieve productivity with it.

> *The next step towards inserting the reengineering technology that we have developed is to transform the RET from a laboratory prototype to a production tool that avionics software maintainers will evaluate on mission essential/critical applications.*

We have already identified some limitations of the RET prototype in Section 7.2, and we recommend addressing these before the Beta test. We also recommend preparing the RET for the test by improving or adding capabilities as summarized in Table 10 and detailed in Appendix B.

The recommendations summarized in Table 10 include three types of improvements. 1) Corrective changes are indicated by differences between the anticipated functionality and actual behavior of the RET prototype that we encountered during Task 4. 2) Perfective modifications cohere the RET software design, e.g., by reimplementing some experimental capabilities that evolved during the project and taking advantage of hindsight. 3) True extensions to the RET represent revelations or observations resulting from our research that the few experience reports available on the techniques identified in Task 1 did not portend.

68

Table 10   RET Enhancements

Install Software Refinery Version 4.0
Support I/O statements
Recognize PARAMETER statements
Save intermediate analysis data to disk
Enhance DFD clustering
Eliminate DFD anomalies
Ada parameter modes
Graph layout
Improve hyperlink performance
DFD editing
Ada pretty-printer
Error reporting
Multi-dimensional character arrays
Packager algorithm
DFD repository nodes
FORTRAN DFD
Packager object ordering
Enhance type deduction
Subprogram visibility
Clip DFD labels
Performance

The problem of quantifying the results of the experiment remains open. Fiscal reality and theoretical defects preclude a fly-off that involves comparing the actual development and subsequent maintenance cost of two versions of the reengineered application. On the other hand, it is difficult to identify a set of software quality metrics with demonstrable predictive power. Much to the chagrin of software producers and consumers alike, an element of mystery remains, and it confounds attempts to devise comprehensive, formulaic, and constructive definitions of "good" software.

Software quality metrics may be used to predict changes in software quality over time within a given organization and environment and may thus form the basis for valid internal quality controls, but they fail to provide instantaneous measurements on any universal, absolute scale of software quality. The implication is that any known metrics computed for two arbitrary software systems can not be used to reliably predict which one will suffer a greater defect rate or cost an arbitrary organization more to maintain.

We may have to make do with inferences drawn from experience reports on the experiment by competent software engineers familiar with the application. The target system that they produce with the RET must submit to whatever subjective or objective measures of software quality the maintainers previously applied to the subject system. The maintainers must *interpret* the results to determine the RET's effectiveness within their organization because the subject and target system environments are different; the metrics are invalid out of context. In short, we must rely on their judgement.

The RET was conceived and designed with a cost-based definition of quality in mind, but for some systems measures of quality derived from, say, formal specifications and proofs of correctless are more apt. We must accept the definition of quality established by the organization responsible for the application, provided that the RET is not misapplied. An engineer whose goals are consistent with those of the RET should be the final arbiter and must answer the question: "Would you use the tool again?" Or better yet, "Would you use the products of the tool?"

Based upon our experience with the RET prototype, we would expect a qualified affirmation to both questions. The modification of large computer programs will remain a most difficult undertaking, with or without tools. The quality of a reengineered product will still be related to the level of the reengineering effort, with or without tools. An enhanced RET, however, will increase the *value* of the end product, where value is a function of quality and cost.

The RET prototype will find a niche in reengineering. It relieves the engineer from syntactical minutia, i.e., differences between the source and target programming language syntax, that divert attention from the more important design and implementation decisions requiring human judgement. By concentrating on tasks that are well-suited to automated support, the RET prototype will reduce the resources needed to reengineer avionics support software and will help the human engineer produce a more maintainable system.

APPENDIX A
VENDOR INFORMATION

Table A-1   Vendor Information

VENDOR	CONTACT
Reasoning Systems	Reasoning Systems, Inc. 3260 Hillview Avenue Palo Alto CA 94304 USA Tel. 415-494-6201 Fax. 415-494-8053 e-mail: reasoning@reasoning.com
Polyhedron Software	Polyhedron Software Ltd. Linden House 93 High Street Standlake WITNEY OX8 7RH United Kingdom Tel. (+44) 0865-300579 Fax. (+44) 0865-300232 Compuserve 100013,461
MIT X Consortium	Bob Scheifler MIT X Consortium Laboratory for Computer Science 545 Technology Square Cambridge MA 02139 USA
Free Software Foundation	Free Software Foundation 675 Mass Ave. Cambridge MA 02139 USA

APPENDIX B
RECOMMENDED RET ENHANCEMENTS

The RET is a prototype and it needs some improvements before production software maintainers who are not experts in language processing could achieve acceptable productivity levels. This section critiques the RET prototype and recommends some enhancements to it that would transform it from a laboratory prototype into a production tool.

Some of the improvements are indicated by differences between the anticipated functionality and actual behavior of the RET prototype that we encountered during Task 4, but most represent discoveries or observations that are a natural part of the research given the dearth of experience reports on the techniques identified in Task 1.

B.1 HIGH PRIORITY

Install Software Refinery Version 4.0 — This version provides better user interface features and is supported by Reasoning Systems. The installation would require modifying some of the RET source code because Reasoning Systems typically moves some functions between packages with each release as they change from officially unsupported to officially supported status.

Support I/O statements — This affects the RG (Ada code generator), Transformer, and DFD. I/O statements weren't very prominent in the sample of FCR source code that we relied upon during RET prototype development because most of the I/O is performed through memory-mapping. Consequently, they received little attention in Task 3. The RET prototype should be modified to recognize READ and WRITE statements.

Recognize PARAMETER statements — An example is the statement that defines MUX_MASK in the MFD subsystem.

Save intermediate analysis data to disk — The RET prototype takes about an hour to read and analyze the REFINE/FORTRAN analysis output file for the FCR subsystem during initialization. This would only need to be done once if the RET could save and reload the intermediate data.

Currently, the RET prototype must reanalyze the REFINE/FORTRAN file whenever it is restarted instead of just reading the intermediate results from disk. REFINE crashes once in a while under normal use. The ability to restart the RET quickly may mitigate the inconvenience to a point where a one week mean-time-between-failure (MTBF) would not decrease productivity to unacceptable levels.

Enhance DFD clustering — We would like to implement repository clustering by name and support manual editing of the DFD to cluster repositories. The DFDs can be so complicated that they are sometimes of little use. We have already made enhancements to the DFD to reduce clutter, but the most complicated diagrams are still too busy.

Allowing the engineer to group repositories by pointing and clicking the mouse would help. The current method of specifying a mapping of repositories to groups before DFD generation worked well on an experimental basis, but is unsuitable for a production tool. The enhanced RET would provide an interactive user interface for this.

Eliminate DFD anomalies — This includes those documented by Benedusi (Ref. 29).

Ada Parameter Modes — Generate "in" and "out" Ada parameter modes based on a data flow analysis in order to generate a simplified DFD. This would very much improve the quality of information in the DFDs. Using "in out" for all parameter modes produces usable Ada code, but the DFDs are either incorrect or inaccurate depending upon the definition of dataflow selected. The DFDs contain bidirectional arrows where they should be unidirectional. The unnecessary bidirectional arrows are misleading, and they increase the number of repository nodes in a DFD.

Graph Layout — Implement a graph layout algorithm for the DFD. The DFDs can get very large, and they may be regenerated whenever the Ada source code changes — as a result of text editing or after rearranging via the Packager. Arranging the DFDs by pointing and clicking with the mouse is tolerable (if you can get an assistant to do it for you) on an experimental basis, and will probably still be necessary to clean up or "fine-tune" an automatically-arranged graph, but repeated manual alteration of graphs that are substantially the same from one generation to the next is intolerable.

In Task 1, we identified an algorithm (Ref. 44) to format an entire graph, or to support interactive graph editing operations that create or delete nodes or edges. Dr. Roberto Tamassia implemented the algorithm in a tool called GIOTTO, and told us that it would be available for a "nominal" grant.

Improve Hyperlink Performance — We implemented hyperlinks (as defined by Reasoning Systems) between several of the RET prototype views, but they were too slow for large subprograms. In fact, they were so slow that we turned off hyperlinking. This is unfortunate because they were so helpful.

There is an inherent tradeoff involving the granularity of Abstract Syntax Graph (ASG) subtrees identified by hyperlinks. As smaller and more numerous syntax elements are referenced by hyperlinks, the search time increases. Either the hyperlinks must be less detailed or faster search strategies must be utilized. We feel that a more effective alternative implementation of hyperlinks is possible with existing technologies and justified.

DFD Editing — The DFD is missing some of the editing options provided for the Packager, e.g., link routing. The DFD should have the same options available as the Packager and the menu items should appear in the same order (to the extent possible).

Ada Pretty-Printer — The RET prototype writes Ada source code to files and then parses and analyzes the code in order to produce the DFD. The RET calls the REFINE/Ada printer to produce the Ada files from the Ada ASGs. The printer doesn't usually format the code in an esthetically pleasing manner. It apparently does not globally optimize the insertion of "tab," "space," and "newline" characters so as to enhance readability.

An Ada pretty-printer should be bundled with the RET prototype. It would not be necessary to integrate the pretty-printer with the RET; it could be invoked to update the source code files after the RET generates them but before it parses them. Many such pretty printers are commercially available or have been developed on government projects.

B.2 MEDIUM PRIORITY

Error Reporting — We spent a great deal of time correcting the transformation rules so that REFINE/Ada would parse and accept all of the Ada code that the code generator produces. We succeeded for most of the subsystems in the Block 40 code, but some problems remain. We suspect that in a rule-based system such as the RET prototype, it will be very difficult to guarantee that the Transformer will work for an arbitrary program.

As a practical matter, the engineer can always edit the generated code and fix the error. The problem is typically very easy for a human to resolve. To facilitate these kinds of corrections until the code generator is made more robust, the RET prototype should report the REFINE/Ada error messages. We wrote some functions to print out the messages and associate them with the source code, but the capability should be integrated with the RET user interface.

Multidimensional Character Arrays — The RET prototype transforms FORTRAN character array variables, such as those in the UFC subsystem, into Ada variables of type "string." This solution doesn't support multidimensional character arrays. The RET could define a family of array types with elements of type character. This approach requires defining some of the standard operations as well.

Packager Algorithm — The RET prototype supports clustering by interconnection strength and by common clients and suppliers, both of which are effective. During Task 4, we noticed many instances in the FCR code where the subprogram names suggest groups, and we clustered them by hand (pointing and clicking on each subprogram) in these cases. The RET prototype should support Packager clustering by name.

DFD Repository Nodes — Benedusi defines both data store and buffer nodes in Hierarchical Data Flow Diagrams to describe two kinds of repositories. Data store nodes represent external data such as files and buffer nodes correspond to data objects such as variables. The RET prototype supports only buffer nodes because of the rarity of file I/O in the FCR code. The RET should support repository nodes if it is to be used on subject systems that perform traditional file I/O (as opposed to memory-mapped I/O).

FORTRAN DFD — The RET prototype generates the DFD view from the generated Ada source code. A DFD generated from the FORTRAN system may be useful during packaging. There are no technical barriers to this, but it isn't clear that a LHS DFD is needed because the Packager contains some information on data flow. The Packager shows data bindings, and the edge labels show the direction of data flow. On the other hand, the direction of data movement is not as obvious in the Packager view as it is in the DFD view. The need for an LHS DFD is an open question.

Packager Object Ordering — This is purely a technical issue. The pak-node-objects attribute is defined in the Packager domain model as a set, but it should be a sequence. The result is that data objects may not be generated in the correct order. This results in Ada code that sometimes doesn't compile when objects are used to initialize other objects. This was not resolved in the RET prototype because 1) there's only one constant (viz. PIO2) in the FCR code that suffers from this, 2) the ripple effect from the change is very large, and 3) work-around is very simple. The user just needs to edit the source code to correct the order of declarations.

Enhance Type Deduction — The RET prototype deduces Ada data types for variables and constants based on the corresponding FORTRAN data types and the way in which the FORTRAN data objects are used. The current implementation for type deduction applies a set of transformation rules that examine all references to each data object. The algorithm makes one pass down each ASG and then one pass up.

This approach is inadequate for some of the more complex cases. The RET *may* benefit from a multicycle, bidirectional technique. Additional research is needed to investigate whether such a technique would be more effective than the existing one, or if it would provide greater value given that the current implementation works most of the time. A new implementation would affect many of the transformations.

The priority of this enhancement depends on how many type conversions are generated. The RET prototype generates a large number of explicit type conversions. The new technique may not significantly reduce the number because Ada has a much stricter type system than FORTRAN.

Subprogram Visibility — The RET prototype generates Ada subprogram (data object) declarations in the package specification or body as needed based on the use of the corresponding FORTRAN subprogram (data object). The RET should allow the engineer to override whether subprograms and data objects are defined in the body or specification. The RET should make an initial assumption based on use, but the engineer should be able to change it. The code generator in the RET prototype places all subprogram declarations in package specifications.

B.3   LOW PRIORITY

Clip DFD Labels — INTERVISTA clips labels on DFD and Packager nodes and edges at the borders of the nodes and text boxes, respectively, in the RET prototype interactive views. When INTERVISTA prints the diagrams, it prints out the entire text of the labels. The labels may be very long, causing the text to overlap the nodes and edges. The RET views are intended as interactive displays and are less effective in hardcopy form, so we did not make this a high priority. The RET prototype should be changed to clip, truncate, or summarize the text.

We don't recommend relying on hardcopies of the RET views instead of the on-line interactive views because the views were designed to be interactive. Much of their power derives from the ability to alter their contents dynamically in order to control the level of detail and the type of information shown.

B-5

Performance — The initial analysis phase of the SCP may benefit from some tuning. We don't know how much performance could be improved because we have never run the profiler, but the initial analysis takes a long time. We could only tune those parts of the SCP that we implemented, not those which REFINE/FORTRAN implements. These performance enhancements would be less important if the RET prototype were capable of saving and reloading the intermediate analysis results.

Data Binding — We decided to add one to the count of data bindings between two subprograms involved in a subprogram call for the purpose of clustering in the Packager. We wanted to produce a package structure with good coupling and cohesion characteristics, but we were not trying to achieve an object-oriented design. We recognized that the RET would not be used to drastically split and recombine subprograms, but would be used to create components based upon functional decomposition, and we wanted to give some weight to the fact that one subprogram called the other.

We don't know if this change produces better clusters because we have never measured this. In any case, the incremented data binding count should not appear in the display because it is confusing. The Packager views for the FCR show many edges that have a data binding count of one, indicating that no data bindings exist between the subprograms, and that one subprogram calls the other. This is distracting when trying to inspect and compare the data binding relationships depicted in a view.

Subprogram Stubs — The RET prototype generates package specifications corresponding to FORTRAN intrinsic functions and external subprograms, i.e., subprograms that are referenced, but not defined. To recognize intrinsic subprograms, the RET prototype maintains a table of all FORTRAN intrinsic function names and the allowable formal parameter types. The particular implementation of this table-driven approach requires multiple table entries for a given subprogram that accepts a variable number of formal parameters. That is, one entry for the subprogram with one argument, another for the subprogram with two arguments, etc.

The implementation is simple, but it doesn't scale well. We have only populated the table with up to three entries for each subprogram. This solution is not elegant, but it covers all cases in the FCR code. The RET prototype could be enhanced to recognize an arbitrary number of actual parameters.

APPENDIX C
ACRONYMS FOR VOLUME I

ALC – Air Logistics Center

ASG – Abstract Syntax Graph

ASRET – Avionics Software Reengineering Technology Project

ASTS – Avionics Software Technology Support Program

CD – Call Diagram (A RET View)

CMU/SEI – Carnegie Mellon University/Software Engineering Institute

CS – Common Client and Supplier Metric

DED – Declaration Diagram (A RET View)

DFD – Data Flow Diagram (A RET View)

DO – Delivery Order

FCR – Fire Control Radar

FSI – File System Interface (A RET Component)

GUI – Graphical User Interface

HA – Hypertext Annotation

HDFD – Hierarchical Data Flow Diagram

IR – Internal Representation

IS – Interconnection Strength

LHS – Left-Hand Side (subject system in RET context)

LOC – Lines of Code

LRM – Language Reference Manual

MSP – Mouse Sensitive Printer (A Component of REFINE)

MTBF – Mean-Time-Between-Failure

OB – Object Base (A RET Component)

OC-ALC – Oklahoma City Air Logistics Center

OFP – Operational Flight Program

PACK – Packager View (A RET View and Component)

PIR – Primary Internal Representation

POB – Persistent Object Base (A Component of REFINE)

PR – Preliminary Restructurer (A RET Component)

RA – REFINE/Ada

RCS – Revision Control System

RES – Restructurer (A RET Component made up of PACK and TRAN Components)

RET – Reengineering Tool

RFT – REFINE/FORTRAN

RG – Representation Generator (A RET Component)

RHS – Right-Hand Side (target system in RET context)

SCL – Source Code Listing (A RET View)

SCP – Source Code Processor (A RET Component)

SDD – Software Design Document

SIR – Secondary Internal Representation

SRE – Software Reengineering Environment

SRS – Software Requirements Specification

ST – Symbol Table

TRAN – Transformer (A RET Component)

UID – User Interface and Display (A RET Component)

WL – Wright Laboratory

WL/AAAF – Avionics Logistics Branch, Wright Laboratory

WL/AAAF-3 – Software Concepts Group, Avionics Logistics Branch, Wright Laboratory

YOYO – You're On Your Own

REFERENCES

1. Corbi, T.A., Program Understanding: Challenge for the 1990s, IBM Systems Journal 28(2), 294–306 (1989).

2. Wilkening, D.E., Kreutzfeld, R.J., and Loyall, J.P., Avionics Software Reengineering Technology (ASRET) Software Reengineering Study Report, Technical Report TR-6661-1, TASC, Reading, Massachusetts, 17 February 1993, to be published as a Wright Laboratory Technical Report.

3. D.E. Wilkening, J.P. Loyall. Software Requirements Specification for the Avionics Software Reengineering Tool (RET) Prototype System, RET-SRS-01. TASC Technical Report TR-6661-2, TASC, Reading, Massachusetts, May 1993, to be published as a Wright Laboratory Technical Report.

4. D.E. Wilkening, J.P. Loyall. Software Design Document for the Avionics Software Reengineering Tool (RET) Prototype System, RET-SDD-01. TASC Technical Report TR-6661-3, TASC, Reading, Massachusetts, August 1993, to be published as a Wright Laboratory Technical Report.

5. D.E. Wilkening, Avionics Software Reengineering Technology (ASRET) Project Final Report, Volume II, TASC Technical Report TR-6661-4, TASC, Reading, Massachusets, 5 May 1995, to be published as a Wright Laboratory Technical Report.

6. Byrne, E.J. and Gustafson, D.A., A Formal Process Model for Software Reengineering: The Analysis Phase, Technical Report TR-CS-91-12, Kansas State University, 12 November 1991.

7. Chikofsky, E.J. and Cross II, J.H., Reverse Engineering and Design Recovery: A Taxonomy, IEEE Software, 13–17 January 1990.

8. Cross II, J.H., Chikofsky, E.J., and May Jr, C.H., Reverse Engineering, Advances in Computers 35, 199–254, 1992.

9. Sittenauer, C., Olsem, M. and Murdock, D., Reengineering Tools Report, Technical Report, Software Technology Support Center (STSC), Hill Air Force Base, Utah, 15 July 1992.

10. Scandura, J.M., Cognitive Approach to Systems Engineering and Reengineering: Integrating New Designs with Old Systems, Software Maintenance: Research and Practice 2, 145–156, 1990.

11. Xinotech Research, Inc., Minneapolis, Minnesota, The Design of the Xinotech Language Translator – Jovial to Ada, Second Revision Edition, Xinotech Technical Report XRI 8911_04.

12. D.E. Wilkening. Avionics Software Reengineering Technologies and Process Model Development. *National Security Industrial Association Software Committee and the Embedded Computing Institute, Naval Air Warfare Center Software Reengineering Workshop Proceedings*, Ridgecrest, California, 12–14 January 1993.

13. D.E. Wilkening, J.P. Loyall, M.J. Pitarys, K. Littlejohn. A Reuse Approach to Computer-assisted Software Reengineering. *Proceedings of the Fourth Systems Reengineering Technology Workshop*, APL Research Center Report RMI-94-003. Monterey, California, 8–10 February 1994.

14. D.E. Wilkening, J.P. Loyall, M.J. Pitarys, K. Littlejohn. An Interactive Reengineering Tool for Constructive Language Translation. *Proceedings From the First Annual Software Engineering Techniques Workshop: Software Reengineering* (Draft), Pittsburgh, Pennsylvania, 8–10 February 1994.

15. D.E. Wilkening, J.P. Loyall, M.J. Pitarys, K. Littlejohn. A Reuse Approach to Software Reengineering, *Journal of Systems and Software*, to appear June 1995.

16. Schwanke, R.W., An Intelligent Tool for Reengineering Software Modularity, Proceedings of the 13th International Conference on Software Engineering, Austin, Texas, 13–17 May, 1991, pp. 83–92.

17. Müller, H.A., Orgun, M.A., Tilley, S.R., and Uhl, J.S., A Reverse Engineering Approach to Subsystem Structure Identification, *Journal of Software Maintenance: Research and Practice*, 5(4), 181–204, December 1993.

18. Hutchens, D. and Basili, V.R., System Structure Analysis: Clustering with Data Bindings, IEEE Transactions on Software Engineering SE-11(8), 749–757, August 1985.

19. Aho, A.V., Sethi, R., and Ullman, J.D., Compilers – Principles, Techniques, and Tools, Addison-Wesley, Reading, MA, 1988.

20. Byrne, E.J., Gustafson, D.A., A Formal Process Model for Software Reengineering: The Analysis Phase, Kansas State University Technical Report TR-CS-91-12, 12 November 1991.

21. Reasoning Systems, *REFINE User's Guide*, 1990.

22. Reasoning Systems, Inc., Palo Alto, CA, INTERVISTA User's Guide.

23. Reasoning Systems, Inc., Palo Alto, CA, DIALEXT User's Guide.

24. Reasoning Systems, Inc., Palo Alto, CA, REFINE/FORTRAN User's Guide.

25. Reasoning Systems, Inc., Palo Alto, CA, REFINE/ADA User's Guide.

26. plusFORT Reference Manual, Revision B, Polyhedron Software Limited, Standlake, Witney, UK, 1993.

27. Scheifler, R.W., X Window System Protocol, Version 11, supplied in machine-readable form on the X Window System distribution tape.

28. GNU Emacs Manual, Seventh Edition, Version 18, Free Software Foundation, September 1992.

29. Benedusi, P., Cimitile, A., and De Carlini, U., A Reverse Engineering Methodology to Reconstruct Hierarchical Data Flow Diagrams for Software Maintenance, *Proceedings of the Conference on Software Maintenance*, Miami, Florida, October 1989, pp. 180–189.

30. Burson, S., Kotik, G.B., and Markosian, L.Z., A Program Transformation Approach to Automating Software Reengineering, *Proceedings of the IEEE Computer Society's International Software and Applications Conference*, 1990, pp. 314–322.

31. VAX FORTRAN Language Reference Manual, Digital Equipment Corporation, Maynard, Massachusetts, Order Number: AA-D034E-TE.

32. Neese, R.E., and German, C.S., "Modular Embedded Computer Software (MECS) Interim Report: Executive Overview," Vol. 1, Technical Report WL-TR-92-1098, Wright Laboratory, Wright-Patterson AFB, OH, 15 April 1991.

33. Neese, R.E., German, C.S., and Giuffre, M.S., "Modular Embedded Computer Software (MECS) Interim Report: Executive Overview," Vol. 2, Technical Report WL-TR-92-1099, Wright Laboratory, Wright-Patterson AFB, OH, 15 April 1991.

34. Blais, R.R. Neese, R.E., and Nohalty, K.L., "Modular Embedded COmputer Software (MECS) Final Report: Executive Overview," Vol. 1, Technical Report WL-TR-92-1100, Wright Laboratory, Wright-Patterson AFB, OH, 15 April 1991.

35. Blais, R.R. Neese, R.E., and Nohalty, K.L., "Modular Embedded COmputer Software (MECS) Final Report: Executive Overview," Vol. 2, Technical Report WL-TR-92-1101, Wright Laboratory, Wright-Patterson AFB, OH, 15 April 1991.

36. Oulsnam, G., Unraveling unstructured programs. *The Computer Journal*. Vol. 25, No. 3, pages 379–387, 1982.

37. Oulsnam, G., The algorithmic transformation of schemas to structured form. *The Computer Journal*, Vol. 30, No. 1, pages 43–53, 1987.

38. Linger, R.C., Mills, H.D., Witt, B.I., *Structured Programming, Theory and Practice*, Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1979.

39. Ada Programming Language. ANSI/MIL-STD-1815A.

40. Sturman, J.N., Achieving software reuse by conversion and reorganization of software systems. *IEEE Proceedings of the National Aerospace and Electronics Conference*. Vol. 2, pages 606–612, 1990.

41. Yau, S.S., A metric of modifiability for software maintenance, *Proceedings of the Conference on Software Maintenance*. Scottsdale, Arizona. IEEE Computer Society Press. Washington, DC, October 1988.

42. Reasoning Systems, *Release Notes for REFINE 3.1 on Sun Computers*, 1991.

43. VAX Ada Language Reference Manual, Digital Equipment Corporation, Maynard, Massachusetts, Order Number: AA-EG29B-TE.

44. Tamassia, R., Battista, G., Batini, C., Automatic Graph Drawing and Readability of Diagrams, *IEEE Transactions on Systems, Man and Cybernetics*, Vol. SE-18, January/February 1989.